SESSION 1

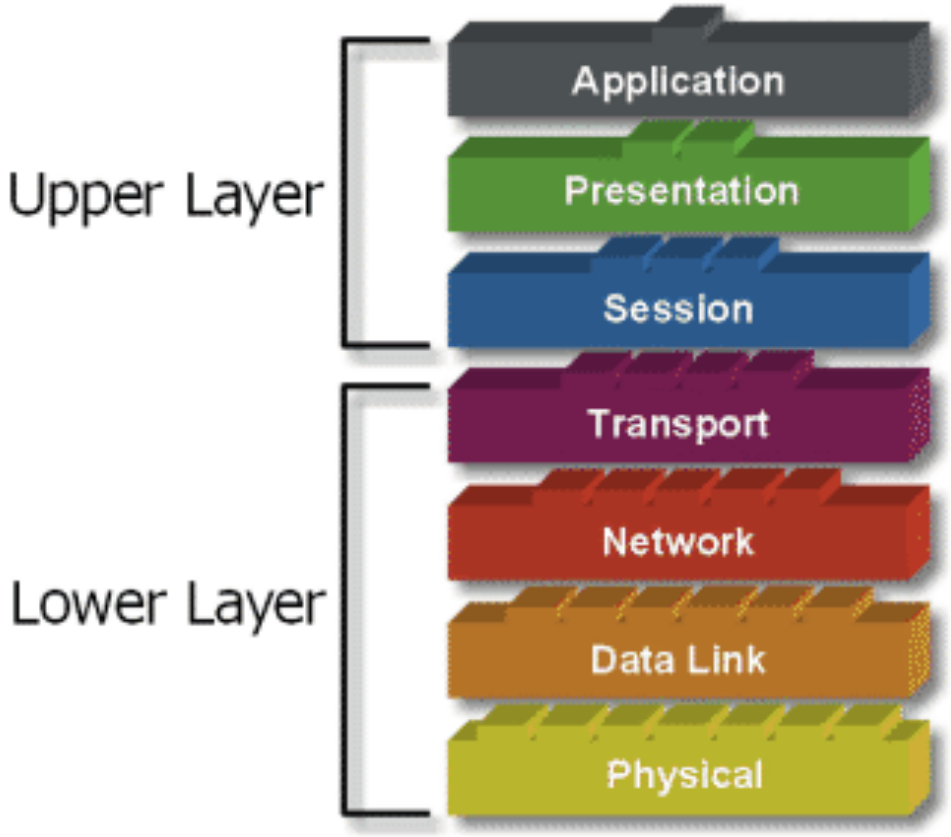
INTRODUCTION TO UNIX/LINUX NETWORK PROGRAMMING

Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

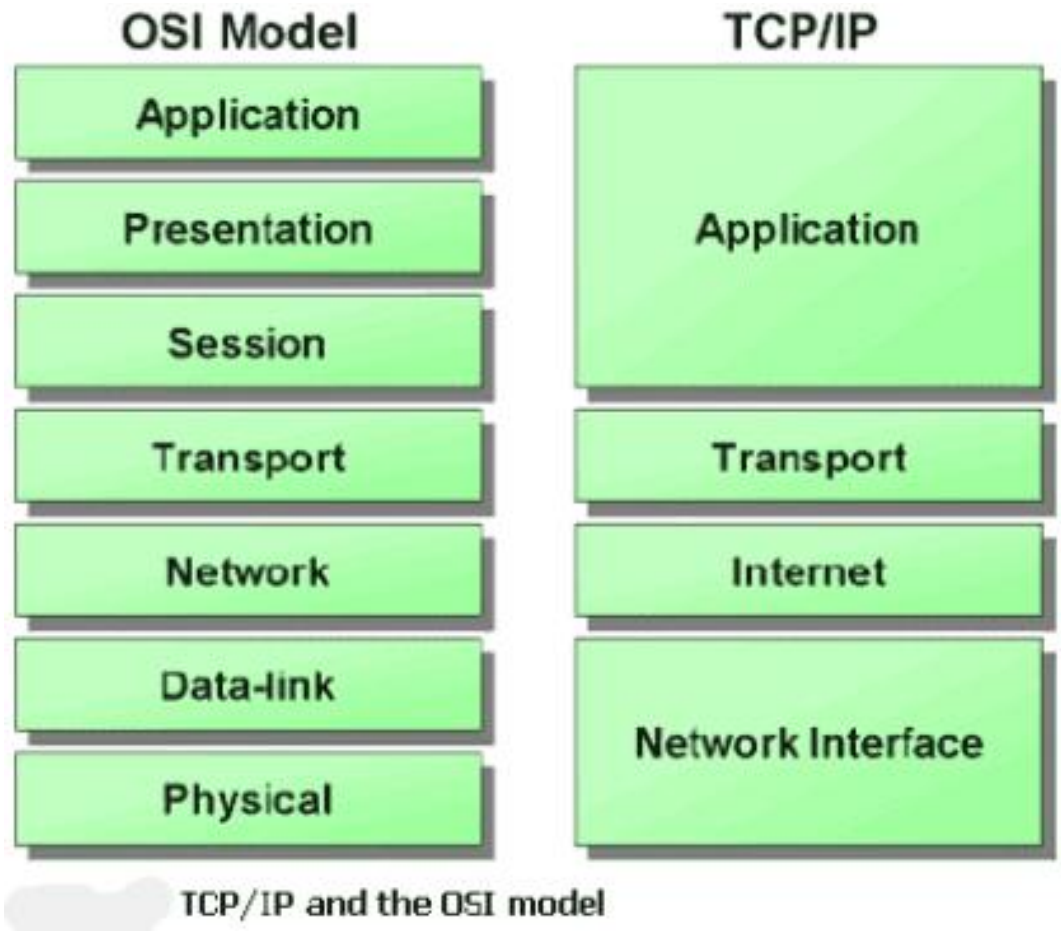
Introduction to Network

- A computer network, often simply referred to as a network, is a collection of hardware components and computers interconnected by communication channels that allow sharing of resources and information
- The Internet is a global system of interconnected computer networks that use the standard Internet protocol suite (TCP/IP)
  - network of networks

ISO OSI Model

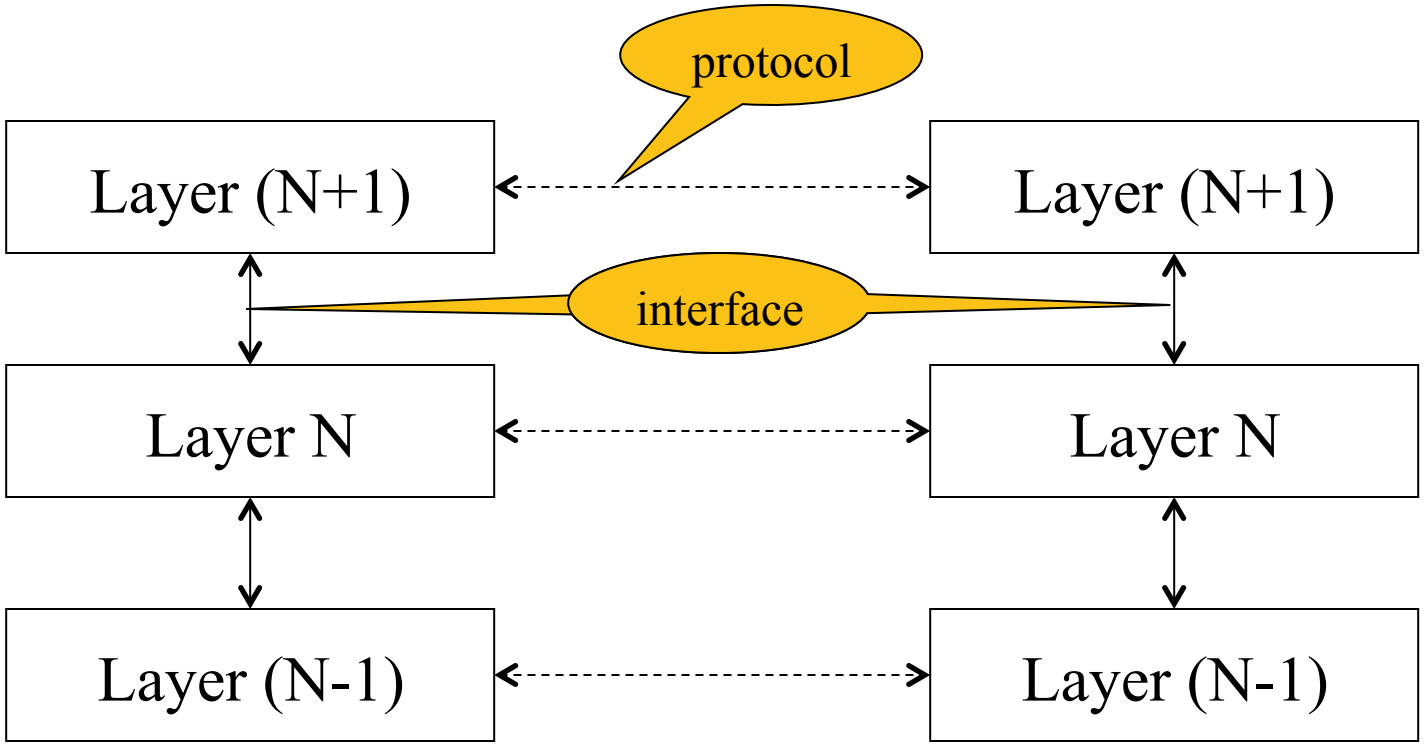


OSI Model and Internet Suite

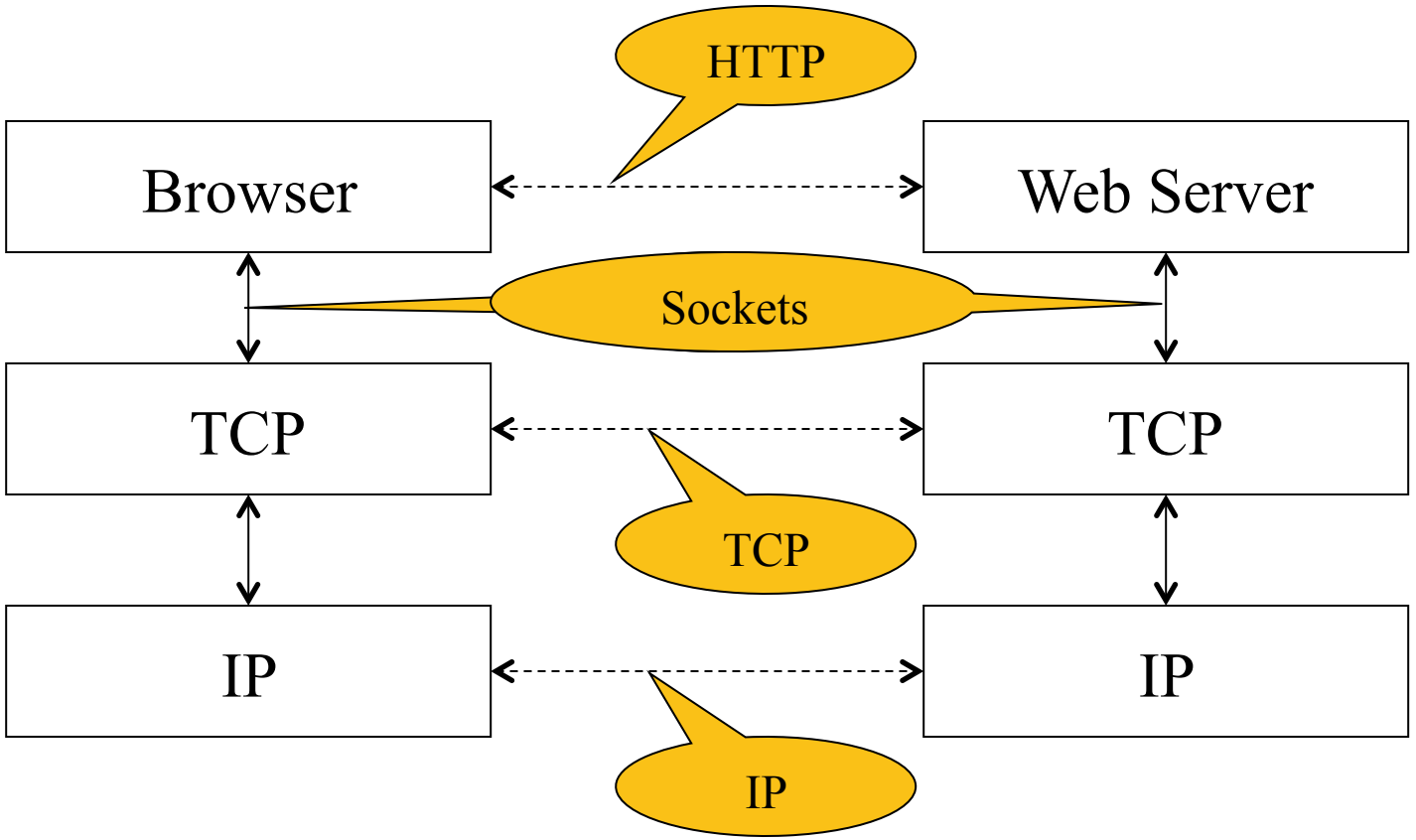


TCP/IP and the OSI model

Protocol and Interface



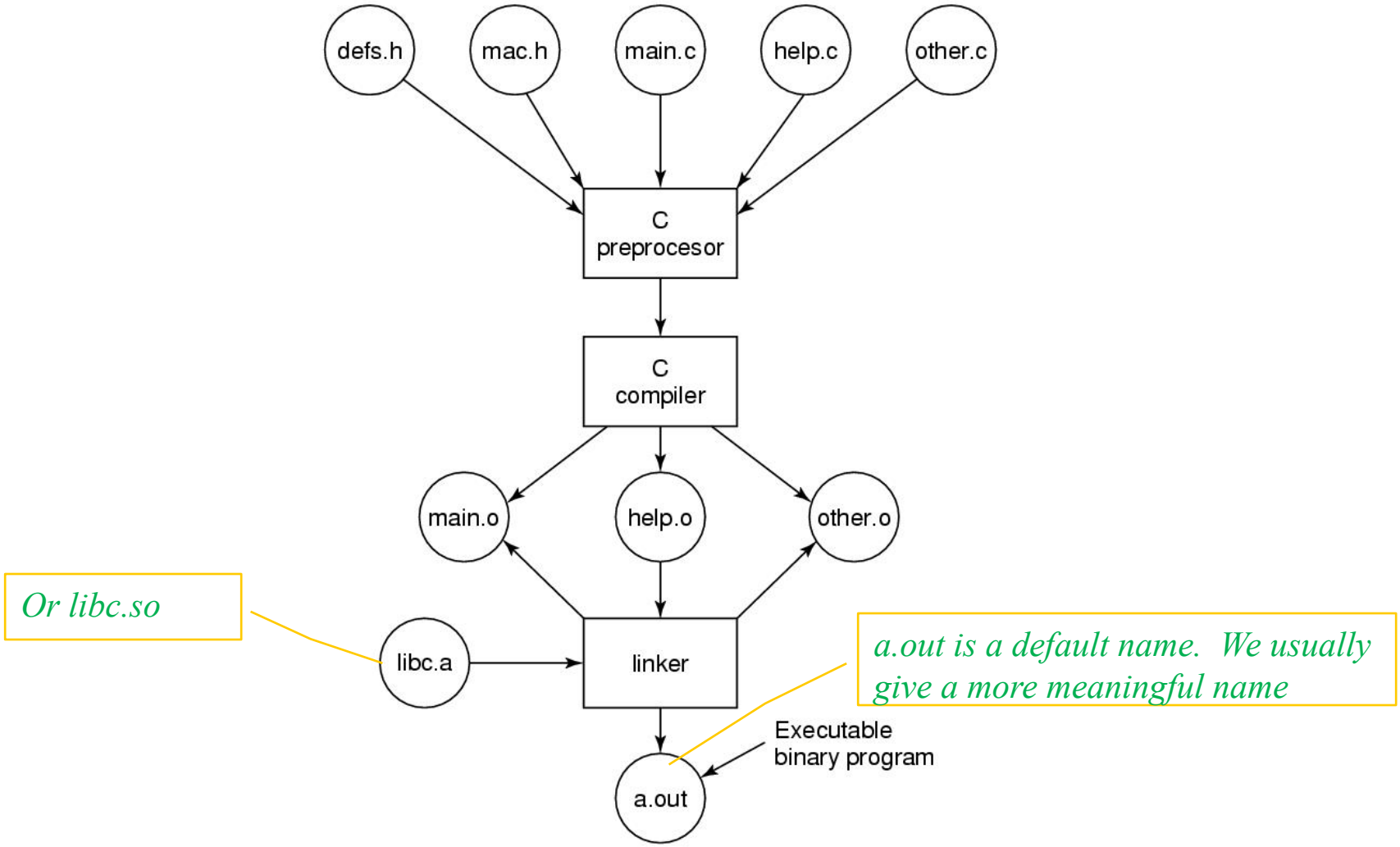
Protocol and Interface Examples



Linux Programming Review

- Tools
  - Compiler
  - Library
  - Debugger
  - Makefile
- Linux Concept
  - Process
  - File I/O

The Compiling and Linking Process



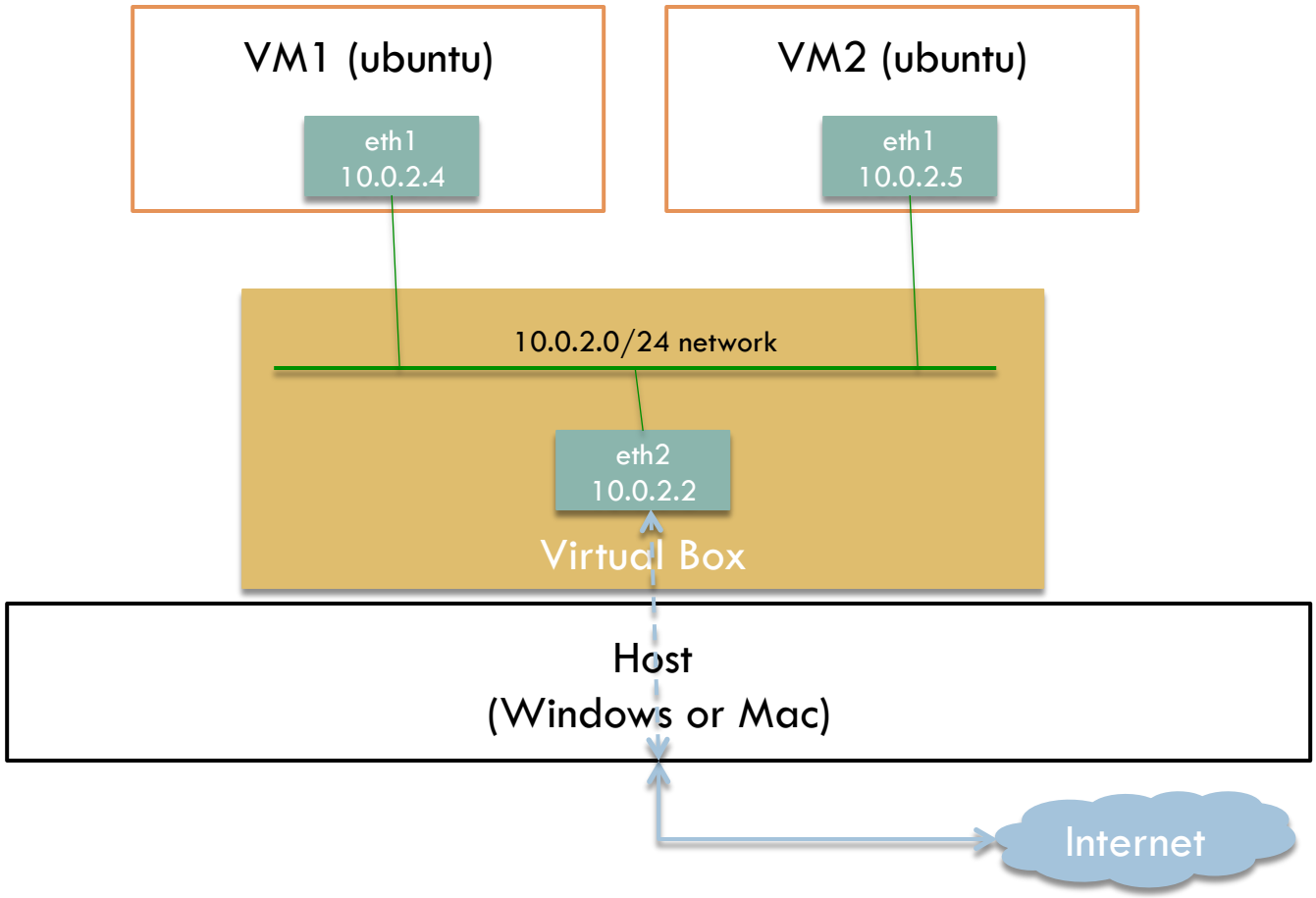
Coding Guideline

- Must indent properly (with tab = 4 spaces)
- Must compile without warnings
- Must use meaningful variable/function/file names
- Must have appropriate comments
- Optionally, use the make system
  - Learn more about make cs515_make_intro.pptx

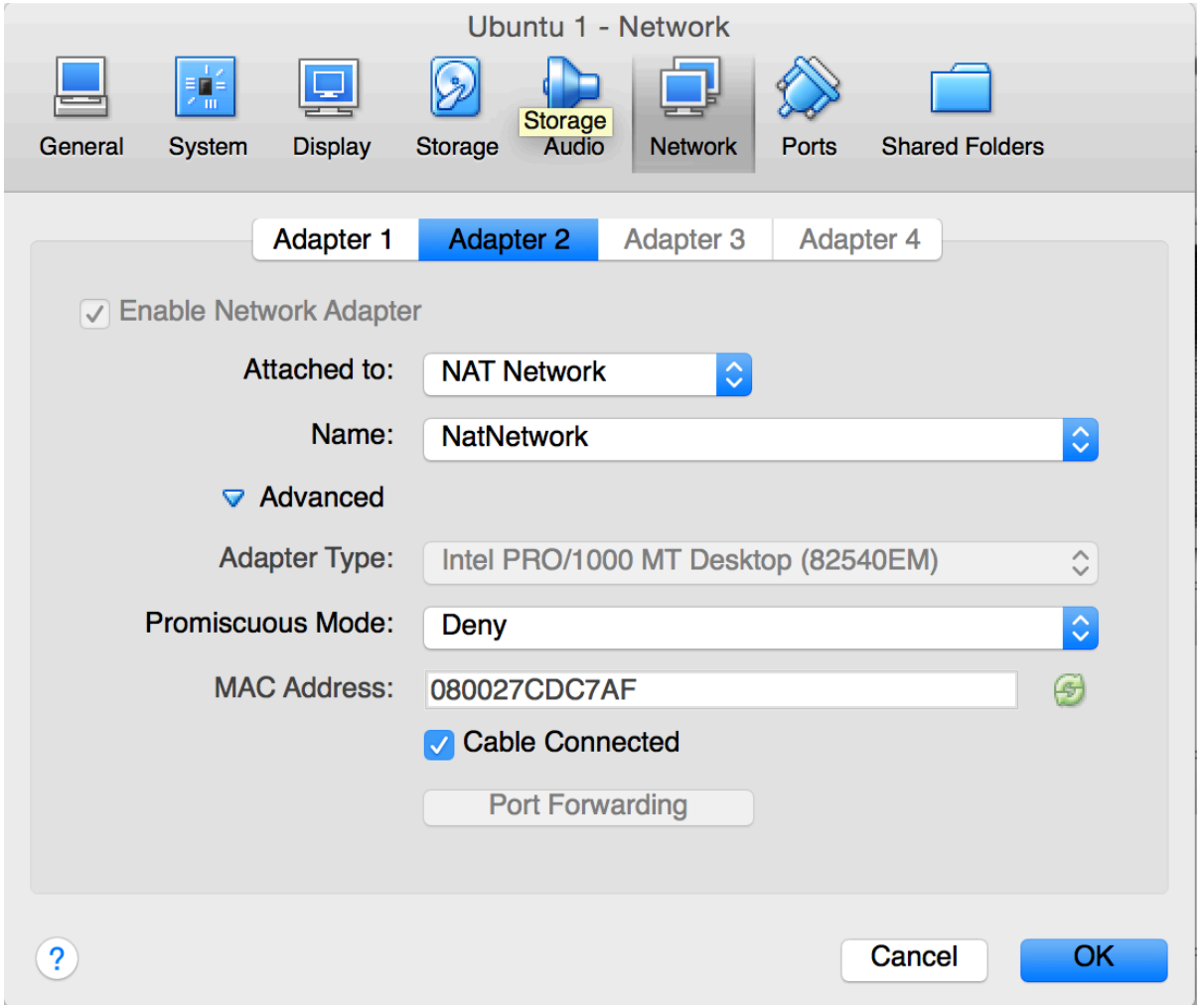
Computer Network Lab

- Create your own computer network lab !
  - Use the lab network you created in CS470
  - If you have not done so, you are highly recommended to create one
    - VirtualBox (https://www.virtualbox.org/)
    - Ubuntu (http://www.ubuntu.com/)
      - Feel free to choose other Linux distributions
    - Instruction (http://www.wikihow.com/Install-Ubuntu-on-VirtualBox)

Virtual Box Nat Network



Configure Nat Network on Virtual Box



NPU Servers

- The selected NPU servers are available for this class
  - npu1, npu8, npu20 …
  - You should already have access to all these servers
- NPU servers vs. VMs
  - The advantage of VM: you can run program as a root with sudo, which allows you to bind to the well-known ports
  - The disadvantage of VM: you probably do not have the host name set up

Text Book Code Examples

- The text book comes with many code examples
  - The examples are related to each other and each example illustrates a concept
  - Section 1.6 Roadmap to Client/Server Examples in the Text
  - Download the source from the text book website

Text Book Code Examples (Cont.)

- Problems with the examples
  - Too many very old style programs
  - Too many wrapper functions for system calls and error handlings
    - The wrapper functions become burdens if you just want to copy some code snippets
  - Lack of coverage of Application Layer protocol in the example

Acronyms

- POSIX, BSD, SVR4
- TCP, UDP, SCTP, IP, ICMP, DNS, RPC
- ISO, OSI
- RFC

SESSION 2
TCP SOCKETS

Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

Socket

- Socket is the programming interface to the Transport Layer
  - Support TCP, UDP and STCP
  - The raw socket also provides programming interface to Routing and other network elements
- Socket is *like* file to applications
  - Socket file descriptor vs file descriptor
  - The file read/write APIs work on sockets with limited features

Overview of TCP Client-server

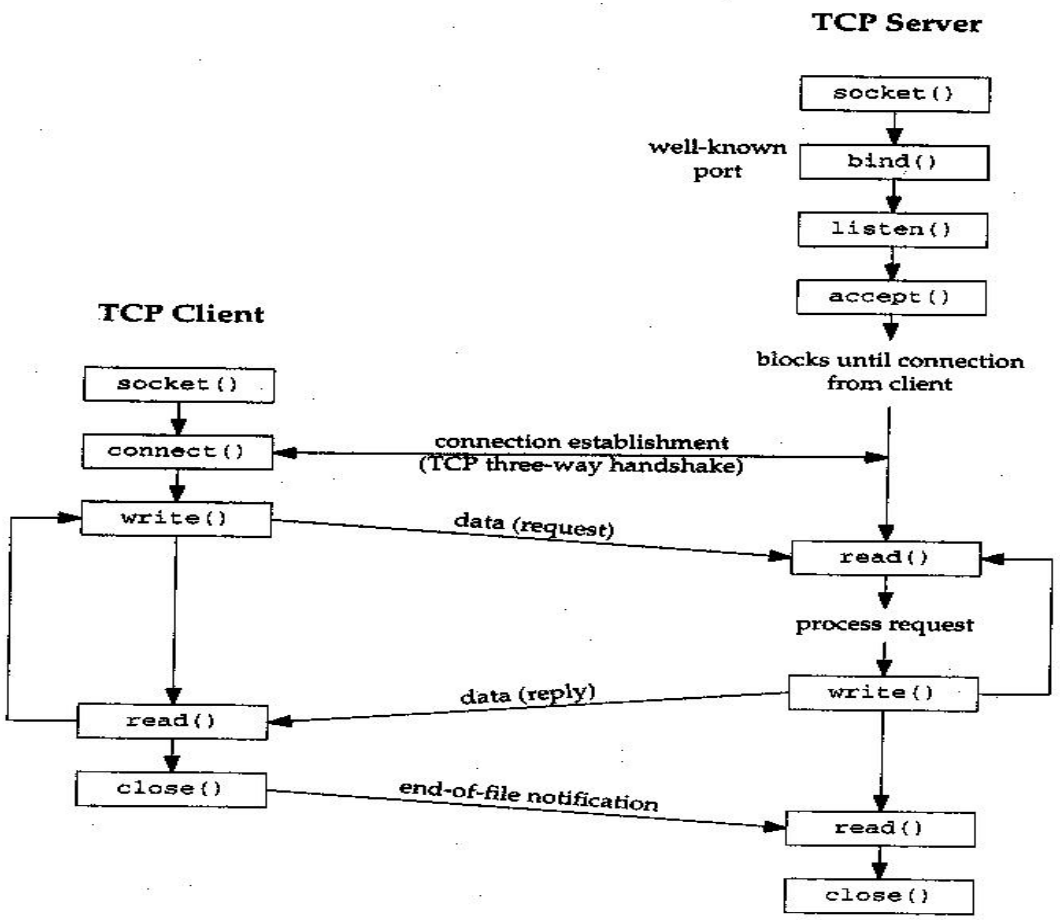


Figure 4.1 Socket functions for elementary TCP client–server.

socket (I)

- socket
  - Function: create an endpoint for communication

```
int socket(int family, int type, int protocol)
```

    - **family**: the communication domain (family) in which a socket is to be created

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

socket (II)

- `type`: the type of socket

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

- `protocol`: an identifier of the protocol that is supported by the address family
  - The default (0) is good enough in most of the cases

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

socket (III)

- combination of family, type and protocol

	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

- return a socket file descriptor (nonnegative integer) if successful, -1 otherwise (error code is set in `errno`)

Socket Address (I)

- Socket address data types
  - Use typedef is a common technique
    - Type `sa_family_t` is a defined in `<sys/socket.h>` as
      ```
      typedef unsigned short sa_family_t
      ```
    - Type `socklen_t` is a typedef of unsigned int
  - Use placeholder structure
    - The `sockaddr` structure is a placeholder for the protocol-specific addresses. The real size of this structure can be bigger

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
```

In the book:
unit8_t sa_len
sa_family_t sa_family

Socket Address (II)

- The Internet (IPv4) address (an Internet-specific address format that cast over **sockaddr**)

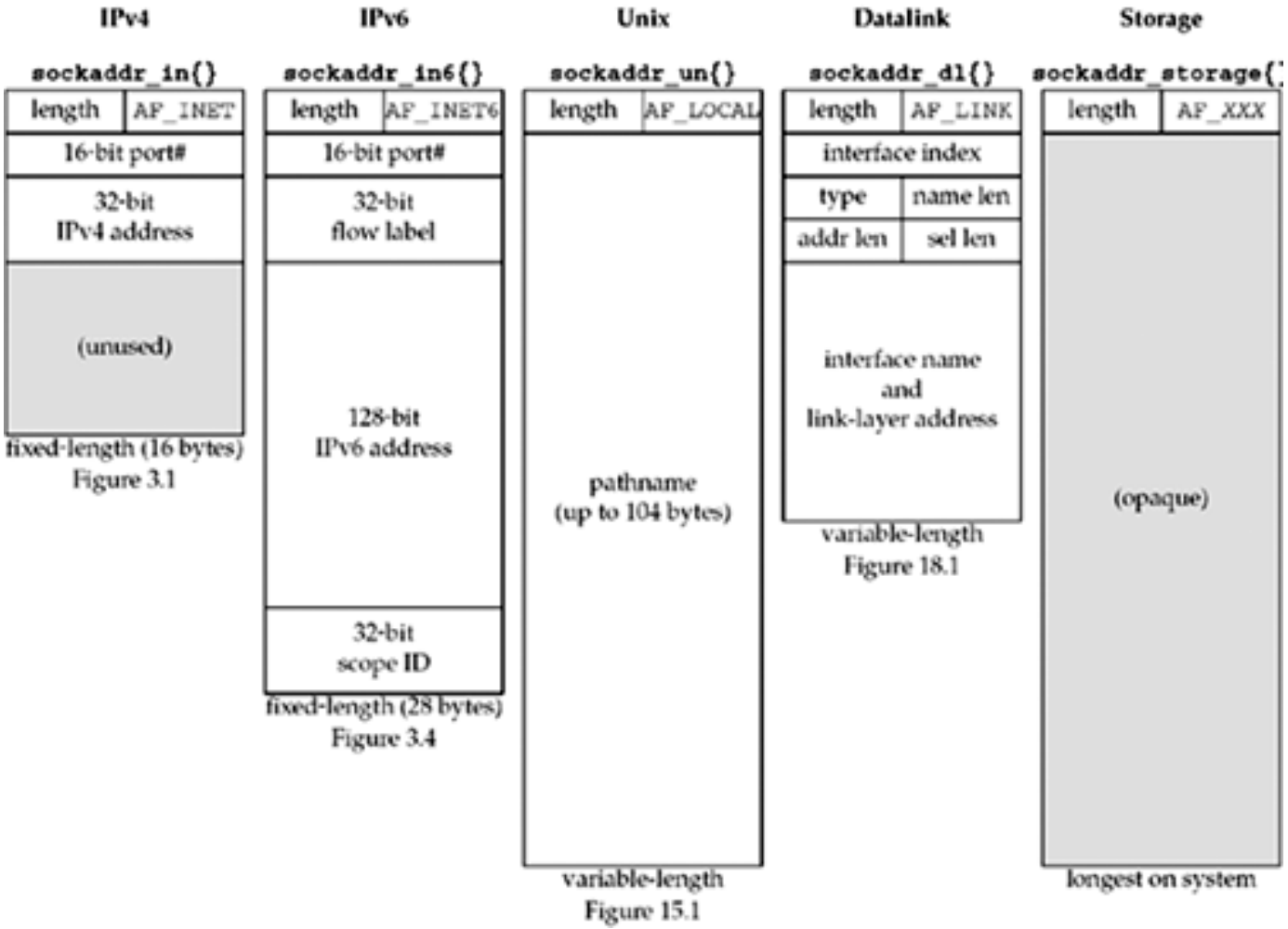
The real definition is always in the header files on your computer. But these header files are really difficult to read!!!!

```
struct sockaddr_in {
   sa_family_t                sin_family;
   in_port_t                  port;
   struct in_addr             sin_addr;
   char                       sin_zero[8];
}
struct in_addr {
   in_addr_t          s_addr;
}
typedef unsigned int          in_addr_t;
```

more padding to support other address types

- Each protocol suite defines its own socket address structure
  - Comparison of various socket address structure (Fig 3.6)

Socket Address (III)



connect

- connect
  - Function: connect a socket (establish a TCP connection)

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

    - **sockfd**: the socket file descriptor
    - **servaddr**: the address of the server to be connected
    - **addrlen**: the length of the address structure pointed by **servaddr** (to support variable length address)
    - return 0 if successful, -1 on error
      - Different **errno** indicates different network problems
- Example: daytimetcpcli

You will find very few daytime servers because it has been replaced by NTP

bind (I)

- bind
  - Function: assign an address to an unnamed socket

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

    - **sockfd**: the socket file descriptor to be bound
    - **myaddr**: the local address to be bound to the socket
      - wildcard
        - IP address: **INADDR_ANY** (IPv4)
        - prot: 0
    - **addrlen**: the length of the address structure pointed by **myaddr**
    - return 0 if successful, -1 on error

bind (II)

- Server usually binds to a well known port
  - Otherwise the client can not find the server
  - Only the super user can bind to the well known port
- Client usually does not bind
  - Client is usually implicitly bound when it calls connect
    - IP address (= host address)
    - Port number (= ephemeral port picked by the kernel)
- An application (server or client) can request the kernel to assign an ephemeral port
  - The application should use `getsockname` to retrieve the port number. Why?

listen (I)

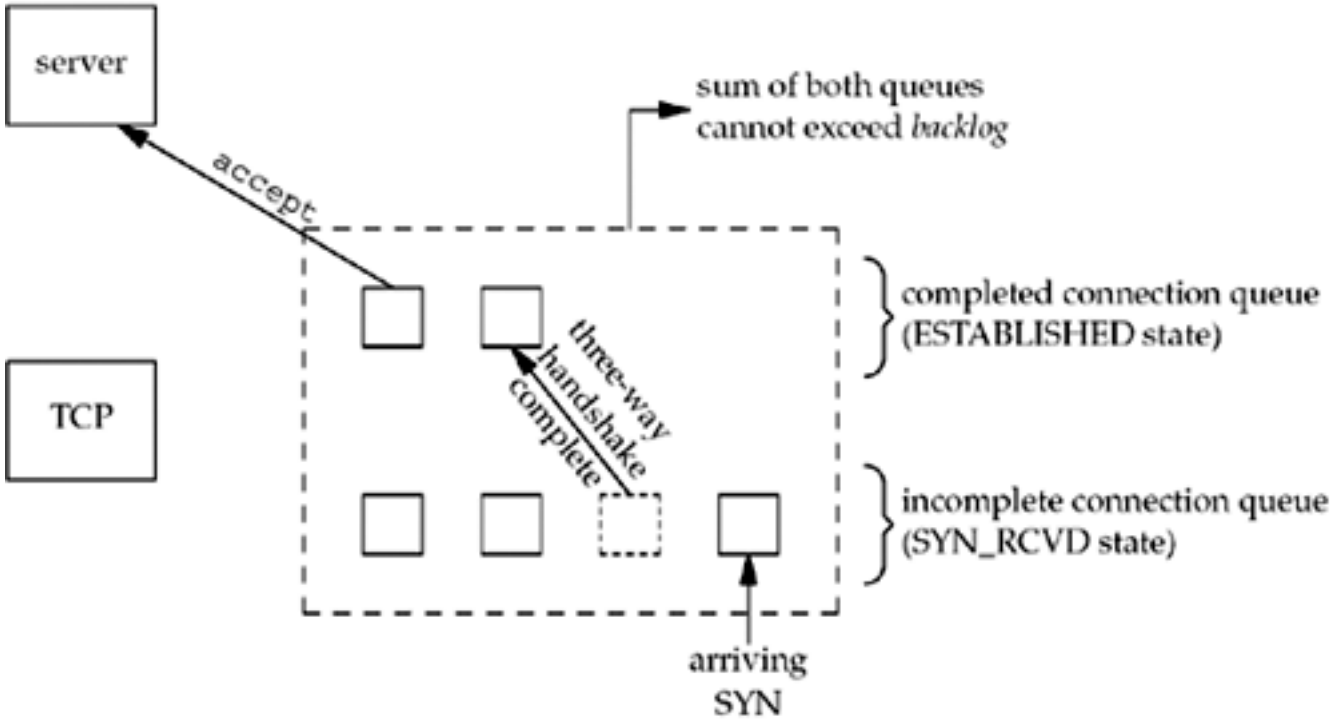
- listen
  - Function: listen for connection on a socket

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

    - `sockfd`: the socket file descriptor to listen on
    - `backlog`: the maximum number of connections that the kernel should queue for this socket
      - connection queue (Fig 4.7)
        - incomplete connection queue
        - completed connection queue
      - was recommended 5 on old UNIX system, now bigger
    - return `0` if successful, `-1` on error

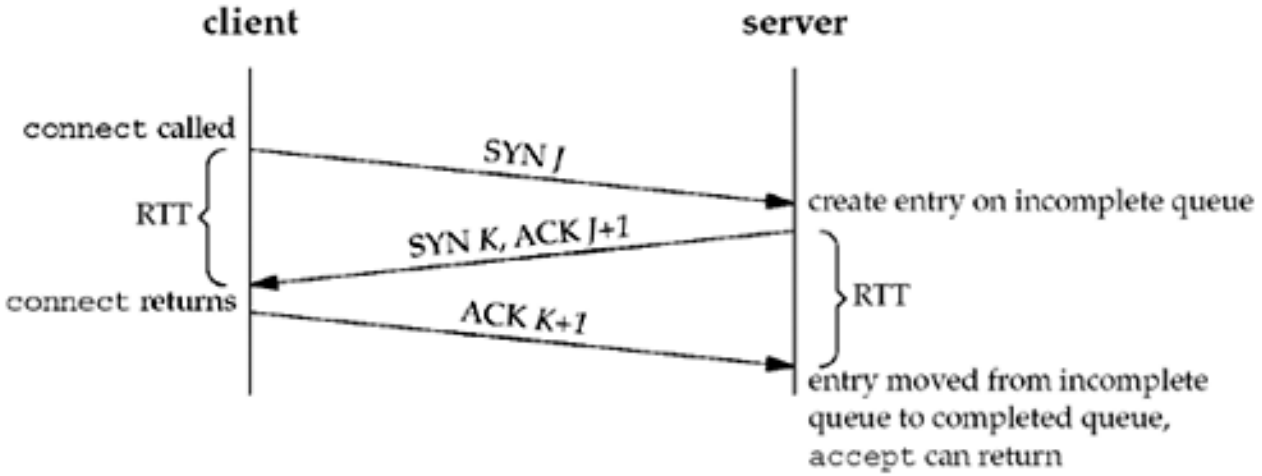
listen (II)

- Two queues maintained by TCP for a listening socket



listen (III)

- TCP three way handshake and the two queues for a listening socket



accept (I)

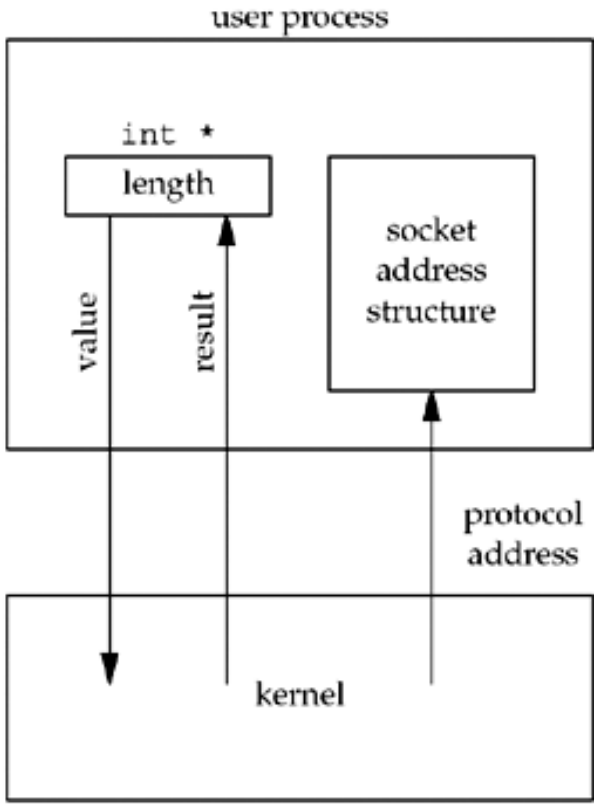
- accept
  - Function: accept a connection request (i.e., return the next completed connection)

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

    - `sockfd`: the socket file descriptor to listen on
    - `cliaddr`: the client address (the other end of the connection)
    - `addrlen`: the size of the `cliaddr`
      - addrlen is a value-result argument
    - return the descriptor (nonnegative integer) for the accepted socket if successful, `-1` on error
  - It is a blocking call

accept (II)

- Socket address structure passed from kernel to process



close

- close
  - Function: close the socket and terminate the TCP connection

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

    - `sockfd`: the socket file descriptor to be closed
    - return `0` if successful, `-1` on error
  - `close` may not start the TCP connection termination procedure if the reference count indicates another process still has the connection

Data Transfer

- The file input and output system calls can be used on socket

```
size_t read(int fd, void *buf, size_t nbytes)
size_t write(int fd, const void *buf, size_t nbytes)
```

- The socket specific system calls

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

- Example: daytimetcpsrv

Example netcalc

- Feature: a simple remote calculator
- Protocol
  - Specified in README file
- Source file organization
- Version 1
  - `netcalc_clnt`
  - `netcalc_srv`

Wrappers or No Wrappers?

- Code with UNP (book) wrappers
  - Example in `netcalc/var0`
- Code with error handling wrappers
  - Example in `netcalc/var1`
- No wrappers
  - Example in `netcalc/var2`

Error Wrappers

- Coding with a simple error wrapper library

  Directory:

  `~bzhang/class/cs515/netcalc/var1/myerr`

  File:

  Head file `myerror.h`

  Library file `libmyerror.a`

Set Up Example netcalc (I)

- Set up the text examples on VM 1
  - Download/copy: `~bzhang/class/cs515/netcalc.tar`
  - Untar: `tar xvf netcalc.tar`
    - The base directory (netcalc) must be in the same directory as unpv13e
  - Read the README file: `netcal/README`
  - Build: `make`
    - To build variations such as var1, var2 and var3, simply change to the directory and issue the make command

Set Up Example netcalc (II)

- Set up the text examples on VM 2
  - Repeat the steps for VM1
- Testing

Server

```
[npu20:/home/NPU_NETLAB/bzhang/class/cs515/netcalc] ./netcalc_srv
**dumpBuf: len  : 6
**dumpBuf: data : 51(Q) 35(5) 2b(+) 36(6) 3d(=) 3d(=)
```

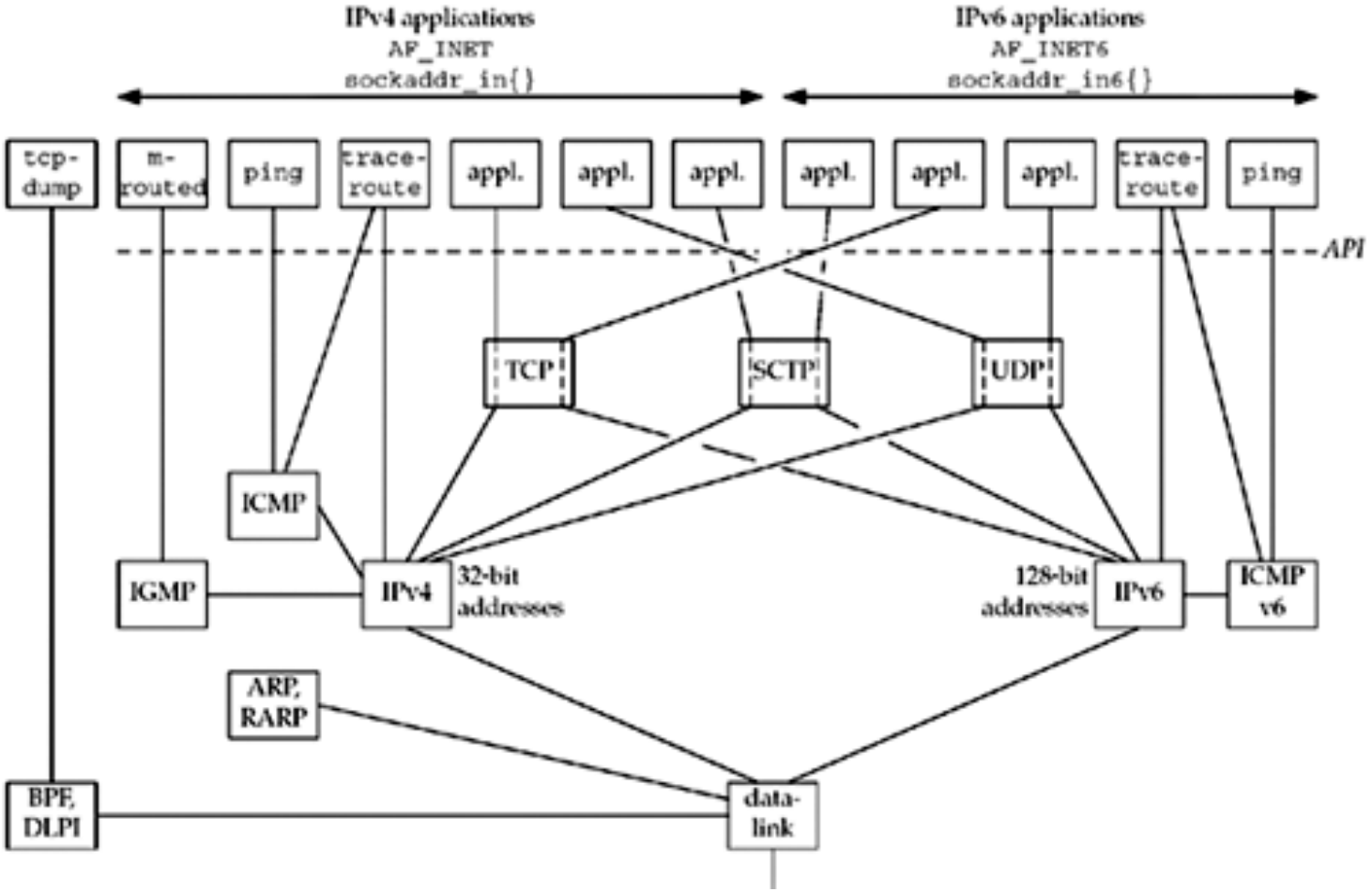
Client

```
[bzhang@npu8 netcalc]$./netcalc_clnt !$
./netcalc_clnt 192.168.0.20
5+6=
Answer: 11
```

SESSION 3
TRANSPORT LAYER PROTOCOLS – TCP, UDP AND SCTP

Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

Overview of TCP/IP Protocols

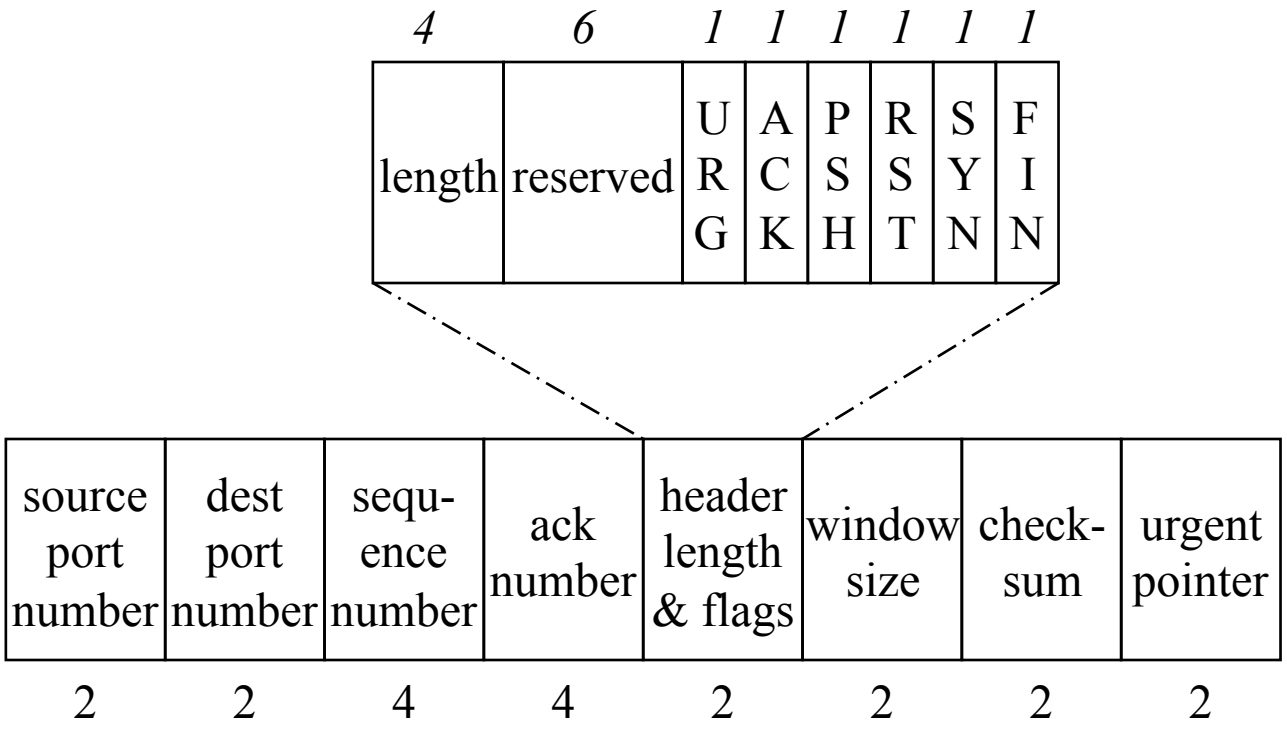


TCP: Transmission Control Protocol

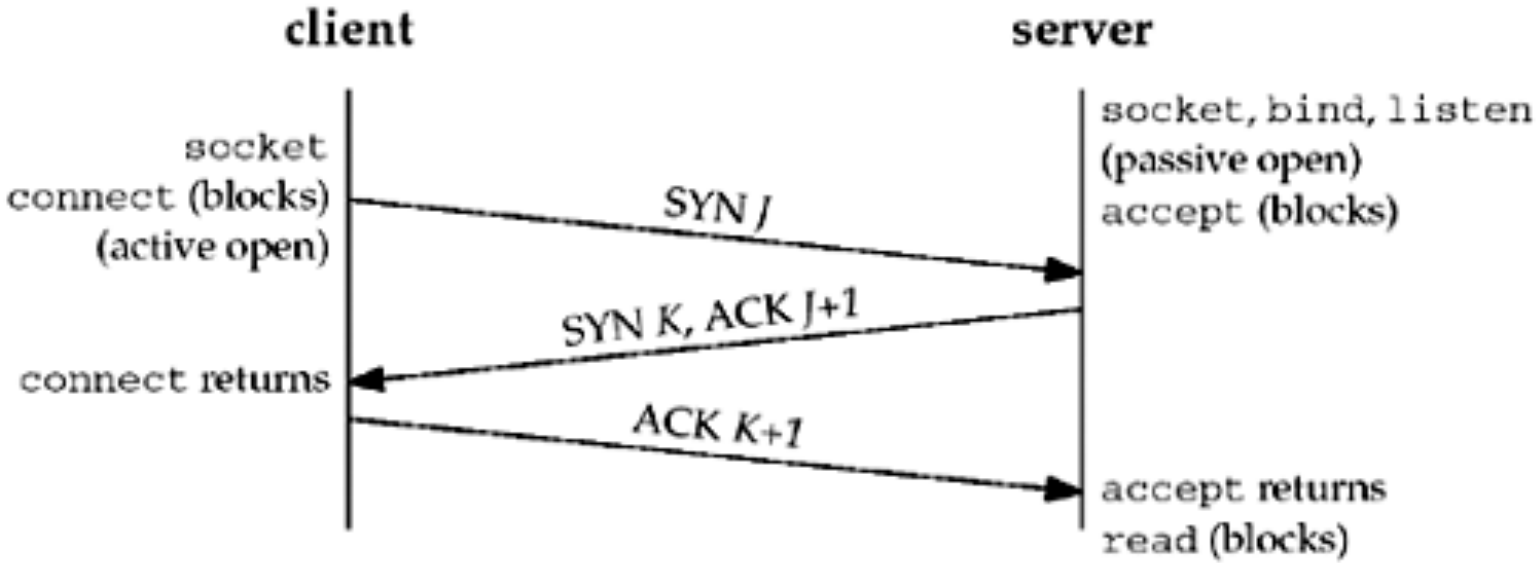
- TCP is the most popular transport layer protocol
  - Connection
  - Reliability
  - Byte streams
    - The unit of data that TCP sends to IP is called a TCP segment
  - Sequence (orderly delivery)
  - Flow control (sliding window)
  - Full-duplex

TCP Header

- TCP header



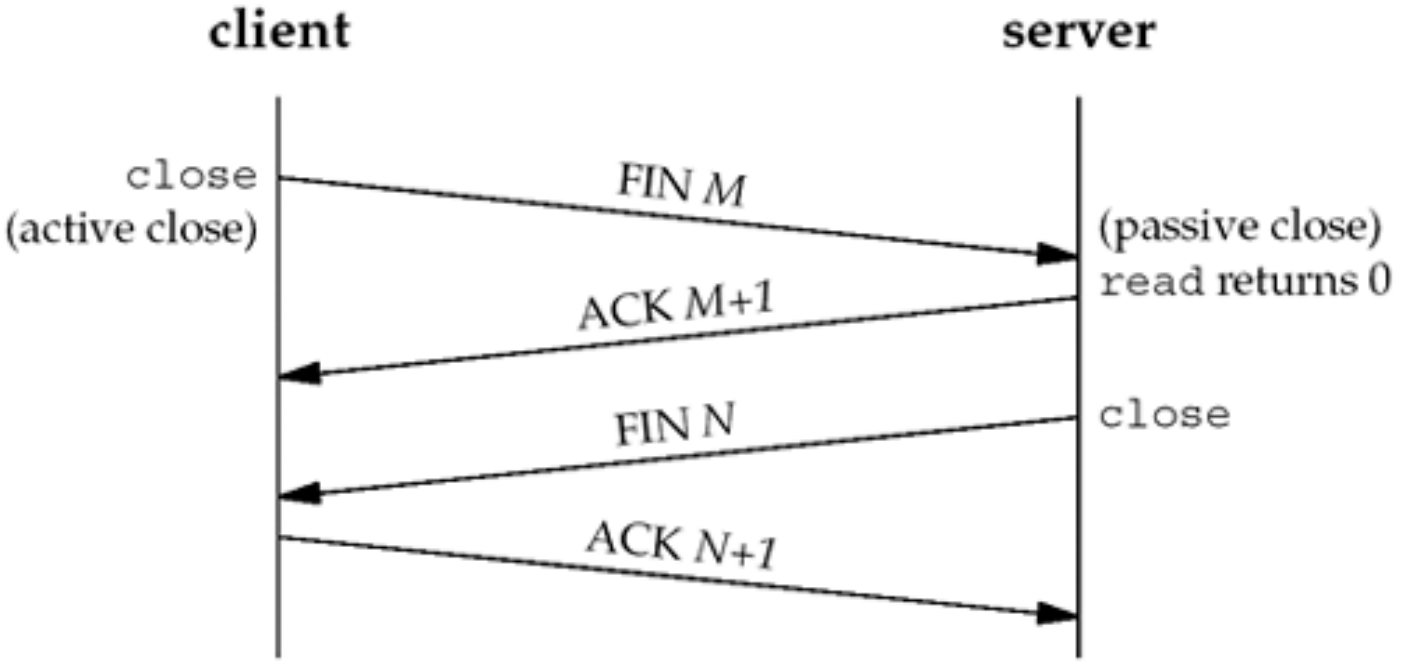
TCP Connection Establishment



The MSS Option

- MSS: Maximum Segment Size
  - Announced by each end to the other end (not a negotiation)
  - Only valid on the SYN segment
  - Calculated from the path MTU (Maximum Transmission Unit)
    - WAN MSS = 536 (576 - 20 - 20)
    - Ethernet MSS = 1460 (1500 - 20 -20)
  - In socket, MSS can be get/set with the `TCP_MAXSEG` option

TCP Connection Termination



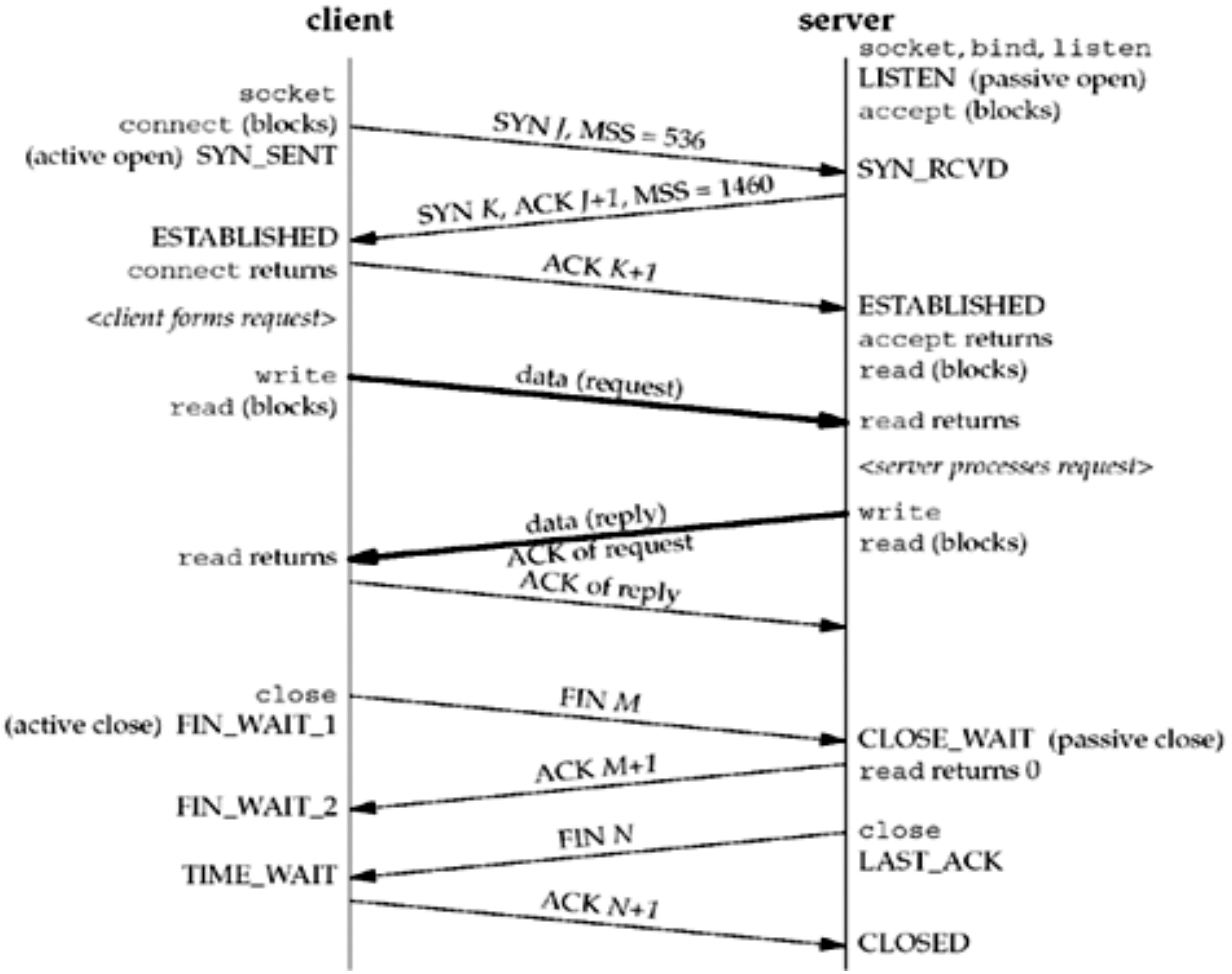
2MSL Wait State

- MSL: Maximum segment lifetime
  - 2MSL is the maximum amount of time any segment can exist in the network before being discarded
  - 2MSL is bounded by IP TTL (Time To Live)
  - 2MSL is chosen by the implementation
- The TIME_WAIT state (also called 2MSL state)
  - The only transition out of this state is the 2WSL timeout event
  - The 4-tuple that defines a connection can not be reused when TCP is in the 2MSL state

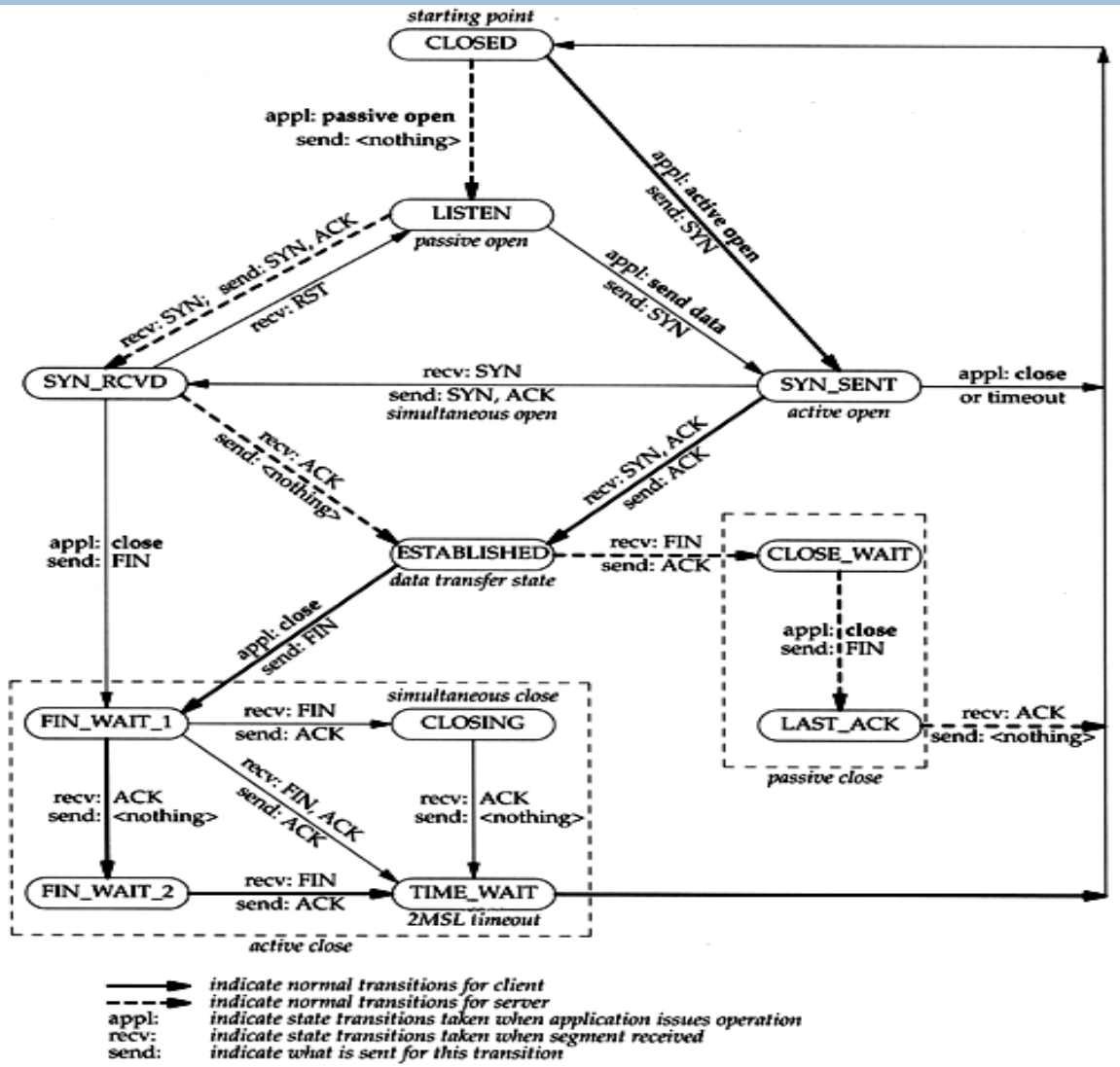
Data Transfer

- Full-duplex data transfer
- Data acknowledgement
  - Delayed acknowledgement
- Sliding-window
  - Offered window (advertised by the receiver)
  - Usable window
  - Maximum window size is not 65536

Data Transfer



TCP State Transition Diagram



UDP: User Datagram Protocol

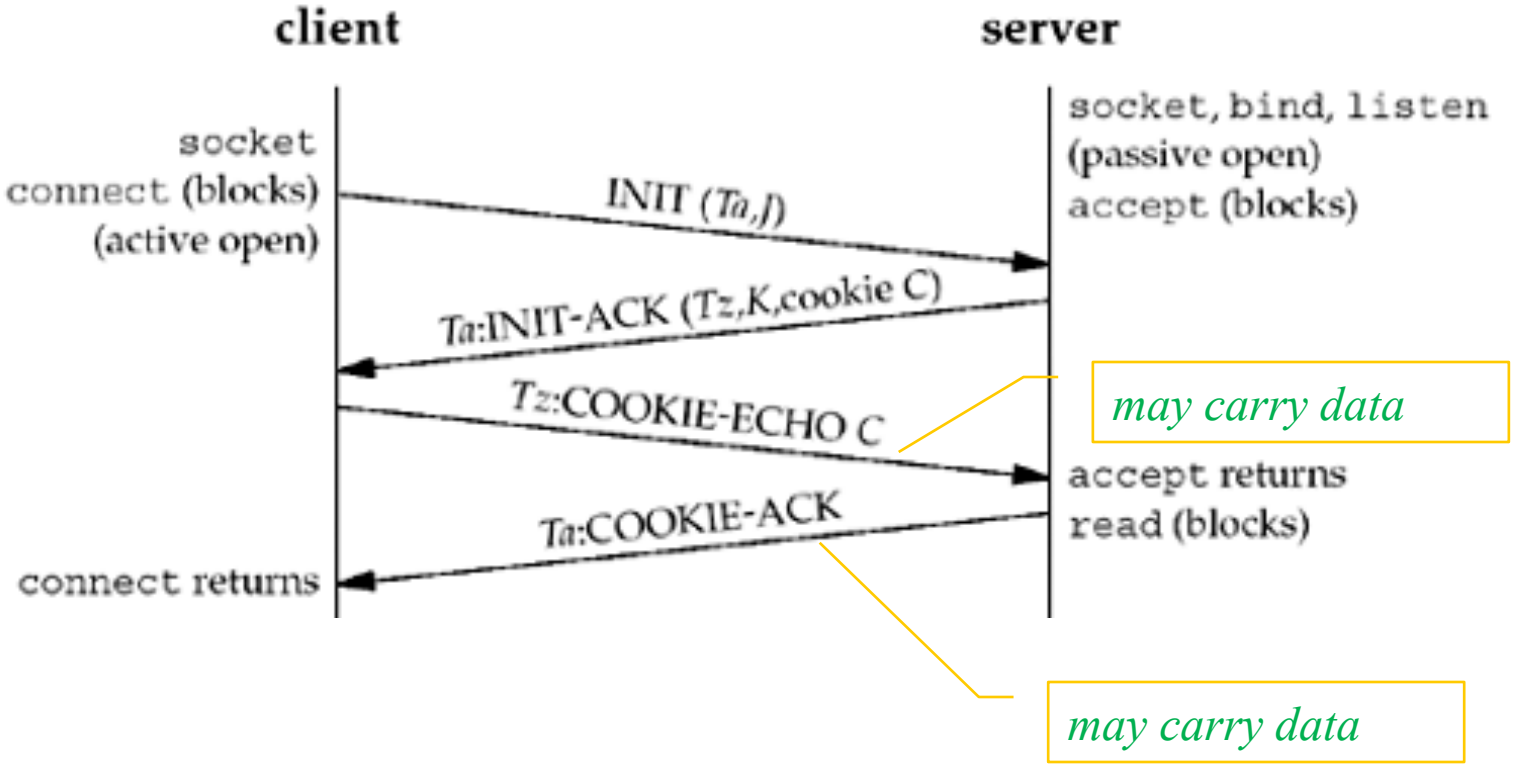
- UDP is a simple transport layer protocol
  - Connectionless
  - No reliability
  - Record-oriented
- UDP header

source port number	dest port number	length	checksum
2	2	2	2

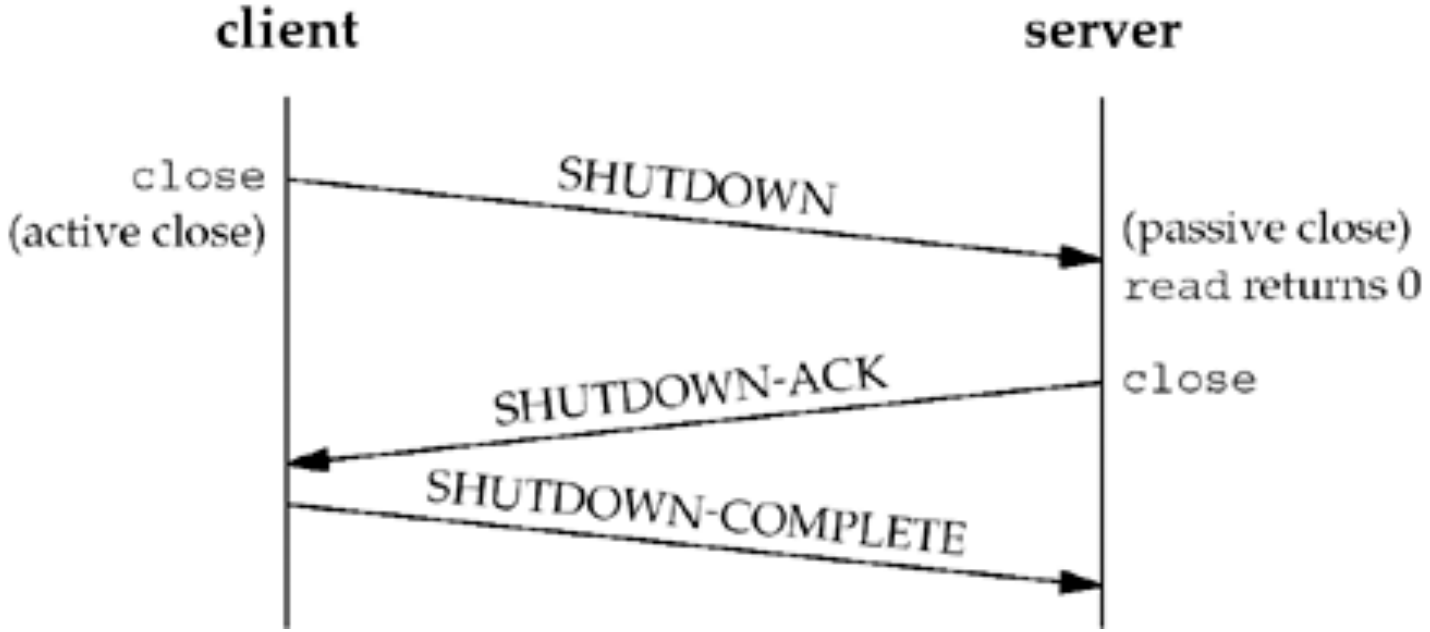
SCTP: Stream Control Transmission Protocol

- SCTP provides associations between clients and servers
  - Like TCP: reliable, in-order deliver, full duplex and flow control
  - Like UDP: message oriented
    - preserve message boundary
  - Additional features
    - multihome support (hence use the term association instead of connection)
    - Multiple streams (avoid HOL blocking)
    - One-to-many support (vs. one-to-one in TCP)

STCP Association Establishment



SCTP Association Termination



Port Number (I)

- TCP/UDP/SCTP uses an abstract destination point called a protocol port
- A protocol port is identified by a 16-bit positive integer (port number)
- Socket
  - Socket = IP address (*incl protocol type*) + port number
  - Uniquely defines an end-point in Internet
- socket pair (4-tuple)
  - Socket pair = source socket + destination socket
  - Uniquely defines a connection in Internet

Port Number (II)

- IANA: Internet Assigned Numbers Authority (http://www.iana.org/assignments/port-numbers)
  - Three ranges of ports
    - Well Known ports: `[0, 1023]`
      - Assigned by IANA
      - Used by system (or root) processes or by programs executed by privileged users
    - Registered ports: `[1024, 49151]`
      - Listed by IANA
      - Used by programs executed by normal users
    - Dynamic and/or private ports: `[49152, 65535]`

In UNIX/Linux, this is also called Reserved Port. Only privileged user can register a port in that range

- The ephemeral ports
  - Used by clients

Common Application Layer Protocols

Application	IP	ICMP	UDP	TCP	SCTP
ping		•			
traceroute		•	•		
OSPF (routing protocol)	•				
RIP (routing protocol)			•		
BGP (routing protocol)				•	
BOOTP (bootstrap protocol)			•		
DHCP (bootstrap protocol)			•		
NTP (time protocol)			•		
TFTP			•		
SNMP (network management)			•		
SMTP (electronic mail)				•	
Telnet (remote login)				•	
SSH (secure remote login)				•	
FTP				•	
HTTP (the Web)				•	
NNTP (network news)				•	
LPR (remote printing)				•	
DNS			•	•	
NFS (network filesystem)			•	•	
Sun RPC			•	•	
DCE RPC			•	•	
IUA (ISDN over IP)					•
M2UA,M3UA (SS7 telephony signaling)					•
H.248 (media gateway control)			•	•	•
H.323 (IP telephony)			•	•	•
SIP (IP telephony)			•	•	•

Useful Programs

- netstat
- ifconfig
  - ip
- ping
- sock
- wireshark
- tcpdump

Review of Homework 1 (I)

- Source code organization
  - Header file `unp.h`
  - Wrapper functions
    - Error handling (`err_sys` etc.)
    - Improved socket library (`Socket` etc.)
- Code quality
  - Many source files are written in *old* UNIX environment
  - Some source files are written in *old* C style

Review of Homework 1 (II)

- Answer the following questions
  1. Where are source files `daytimetcpcli.c` and `daytimetcpsrv.c`?
  2. Function `err_sys` is used on line 14 in `daytimetcpcli.c`. In which file is it defined?
  3. Function `Socket` is used on line 12 in `daytimetcpsrv.c`. In which file is it defined?
  4. Why does the client fail when you run it like this?

     ```
     ./daytimetcpcli 127.0.0.1
     ```

  4. Why does the server fail on the NPU servers when you run like this?

     ```
     ./daytimetcpsrv
     ```

SESSION 4
TCP SOCKETS (II)

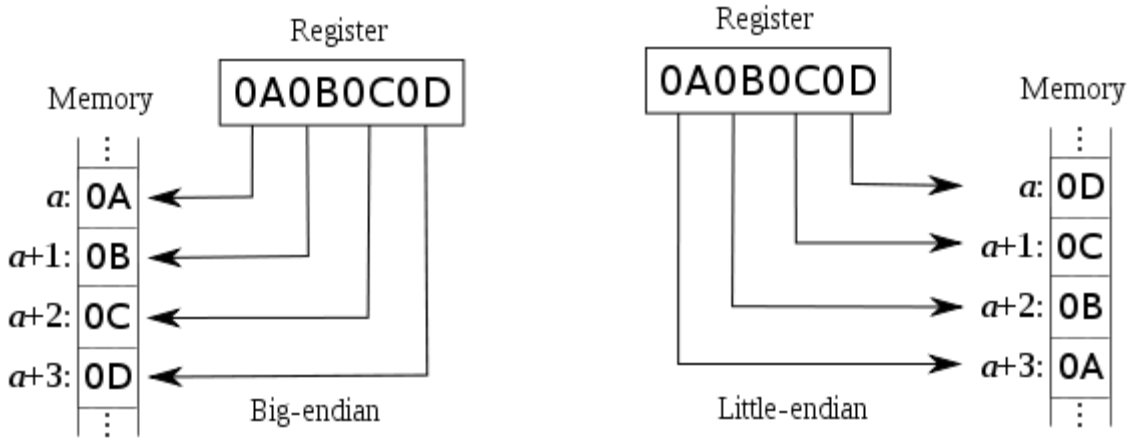
Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

Host Byte Order (I)

- Definition: ways to store multiple bytes (2 or 4 or even more) in the memory
- Two byte orders
  - Big-endian: the high-order byte at the starting (lower) address.
    - PPC and MIPS are big-endian
      - Modern CPUs such as PPC and ARM can be configured to support either byte order
  - Little-endian: the low-order byte at the starting (lower) address.
    - Intel is little-endian

Host Byte Order (II)

- Byte order illustration



- Program to determine the byte order

  `unpv13e/intro/byteorder`

Network Byte Order (I)

- Network byte order
  - The way that multiple byte field (such as the IP address) sent over the network
  - It is in fact a big-endian system
- Application byte order
  - Byte order has always been an issue when we try to share a binary file among different hardware
    - HTML is portable because it is not binary
  - It is up to the Presentation Layer to solve the byte order issue for the Application Layer protocols

Network Byte Order (II)

- Byte order conversion functions
  - These functions are needed for the Transport Layer and Network Layer protocols
- Host to network

```
uint16_t htons(uint16_t host16bitvalue)
uint32_t htonl(uint32_t host32bitvalue)
```

- Network to host

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

Byte Manipulation Functions

- Memory manipulation functions

```
void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t
  nbytes)
void *memmove(void *dest, const void *src, size_t
  nbyes)
int memcmp(const void *ptr1, const void *ptr2, size_t
  nbytes)
```

src and dest may not overlap

src and dest may overlap

- The byte manipulation functions are deprecated

```
bzero = memset, bcopy = memcpy, bcmp = memcmp
```

- Compare them to the string functions

IP Address Conversion Functions (I)

- inet_aton
  - Function: convert IP address from ASCII string to binary

```
int inet_aton(const char *strptr, struct in_addr *addrptr)
```

    - **strptr**: pointer to an IP address string in the dotted-decimal format (e.g., 121.23.55.1)
    - **addrptr**: pointer to an IP address in the 32-bit network byte order
    - return 1 if the string is valid, 0 on error
- A sample implementation is in `libfree/inet_aton.c`

Be careful with files in the libfree directory. If you build it, the APIs are added to libunp.a (not recommended!)

IP Address Conversion Functions (II)

- inet_ntoa
  - Function: convert IP address from binary to ASCII string

```
char *inet_ntoa(struct in_addr inaddr)
```

    - `inaddr`: IP address in the 32-bit network byte order
    - return pointer to the IP address in the dotted-decimal format
  - A test program is `testprog/test_inet_ntoa.c`

IP Address Conversion Functions (III)

- Two more IP address conversion functions

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

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

- They cover both IPv4 and IPv6
- A test program is `unpv13e/libfree/test_inet_pton.c`

Notice that the user provides the return buffer

Socket Information Function

- Socket information functions (get socket address using the socket file descriptor)

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

Actually means address instead of name

gethostbyname (I)

Obsolete in Linux; only for IPv4; replaced by getaddrinfo

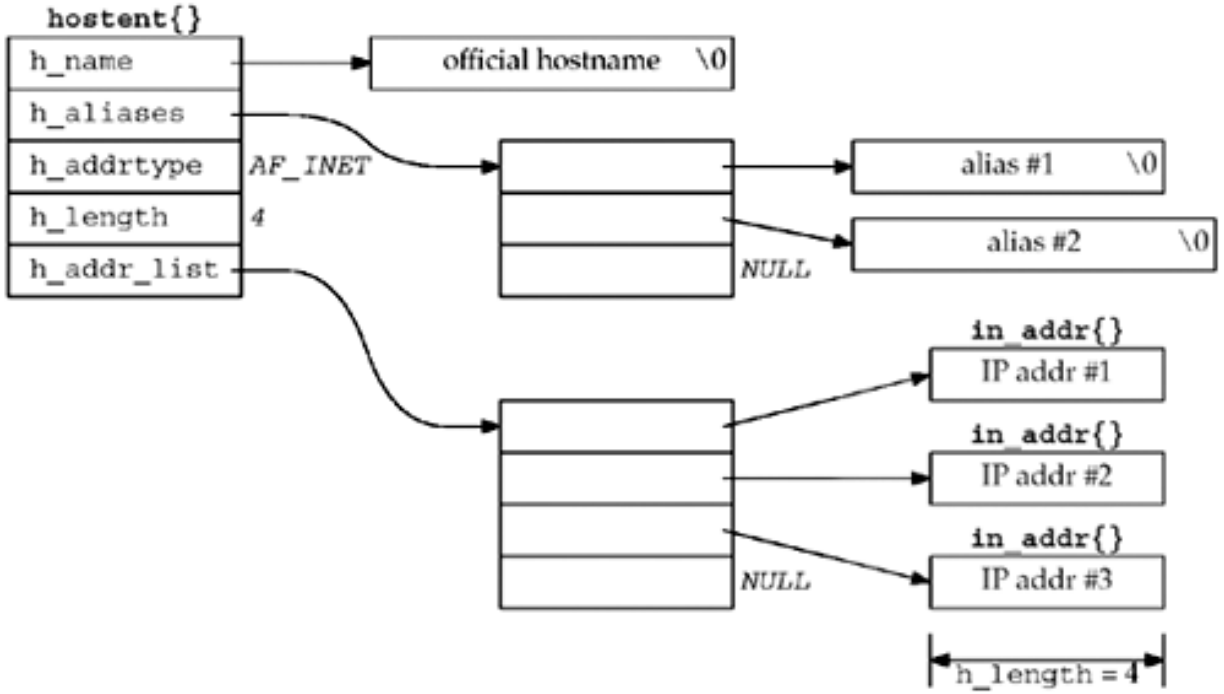
- **gethostbyname**
  - Function: get a pointer to the hostent structure (find the IP address info using the host name)

```
struct hostent *gethostbyname(const char *hohstname)
```

    - **hostname**: the name of the host
    - return the pointer to a **hostent** structure if successful, **NULL** otherwise

```
struct hostent {
   char    *h_name;         /* official name of host */
   char    **h_aliases;     /* alias list */
   int     h_addrtype;      /* address type */
   int     h_length;        /* length of address */
   char    **h_addr_list;   /* list of addresses */
}
```

gethostbyname (II)



- A test program is `names/hostent.c`

gethostbyaddr (I)

Obsolete in Linux; only for IPv4; replaced by getnameinfo

- **`gethostbyaddr`**
  - Function: get a pointer to the hostent structure (find the host name info using the IP address)

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

    - **`addr`**: a pointer to an **`in_addr`** structure (not the IP address in ASCII)
    - **`len`**: the length of the address structure pointed by **`addr`**
    - **`family`**: the address family
    - return the pointer to a **`hostent`** structure if successful, **`NULL`** otherwise

gethostbyaddr (II)

- A test program is `unpv13e/names/hostent2.c`

```
./hostent2 npu1
./hostent2 216.133.192.31
./hostent2 yahoo.com
```

gethostname

- **`gethostname`**
  - Function: get the host name

```
int gethostname(char *name, int namelen)
```

  - **`name`**: the buffer in which the hostname will be stored
  - **`namelen`**: the length of the buffer pointed by name
  - return `0` if successful, `-1` otherwise

Service Information Function

- **getservbyname**

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

```
struct servent {
  char         *s_name; /* service name */
  char **s_aliases;     /* alias list */
  int  s_port;          /* port num, network order*/
  char *s_protocol;     /* protocol to use */
}
```

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

Example netcalc (version 2)

- Version 2
  - `netcalc_clnt_v2`
  - `netcalc_srv_v2`
- File sharing between version 1 and version 2
- Inter-operation between version 1 and version 2

SESSION 5
TCP SOCKETS (III)

Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

Server Design Choice

- Server design
  - Iterative server
    - Multiplex on multiple socket file descriptors
  - Concurrent server
    - Spawn a child server to service a client
- Example: netcalc concurrent server
  - `netcalc_srv_v3`

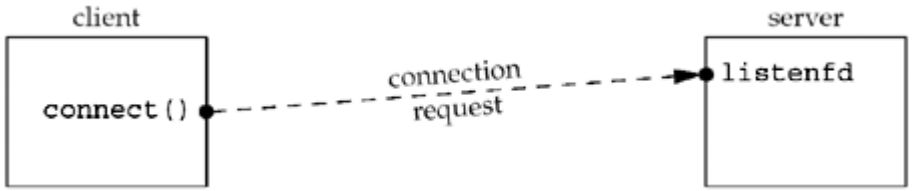
Review of UNIX Process Management

- Fork
- Exec
- Wait
- Signal

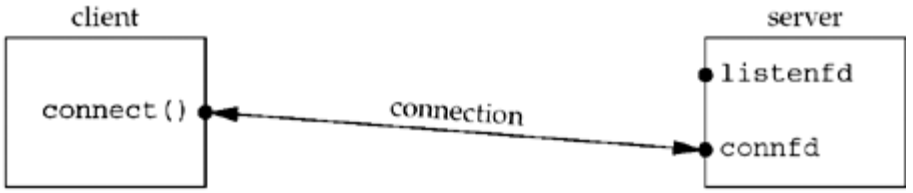
CS506: Advanced UNIX/Linux System Programming

Concurrent Server (I)

- Socket descriptor management
  - Before `accept`

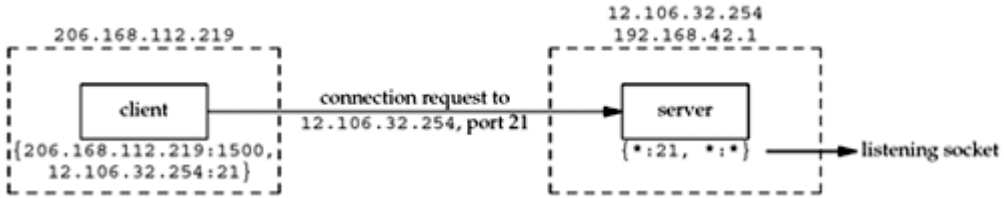


  - After `accept`



Concurrent Server (II)

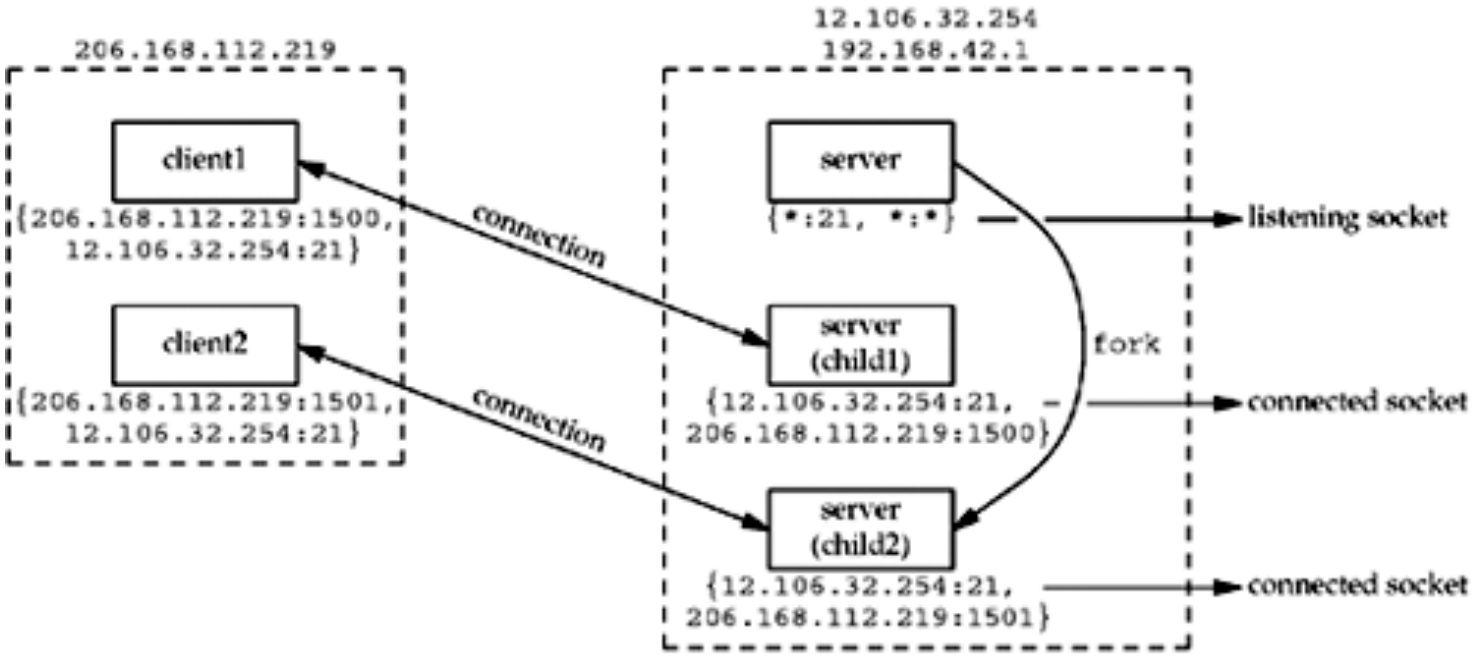
- Port number management
  - Before child



  - After child taking over the connection

Concurrent Server (III)

- After second child taking over the connection



poll (I)

- **poll**
  - Function: input/output multiplexing

```
int poll(struct pollfd *fdarray, nfds_t nfds, int timeout)
```

  - **fdarray**: the file descriptors to be examined and the events of interest for each file descriptor
  - **nfds**: the number of entry in the **fdarray**
  - **timeout**: the number of milliseconds to wait
  - return value
    - >0 : the number of ready file descriptors
    - 0 : timeout
    - <0 : error (-1)

An alternative to poll: `select`

poll (II)

- Events that cause `poll` to return
  - `poll` is a blocking call (the kernel is doing the polling)
  - Events that cause poll to return

Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

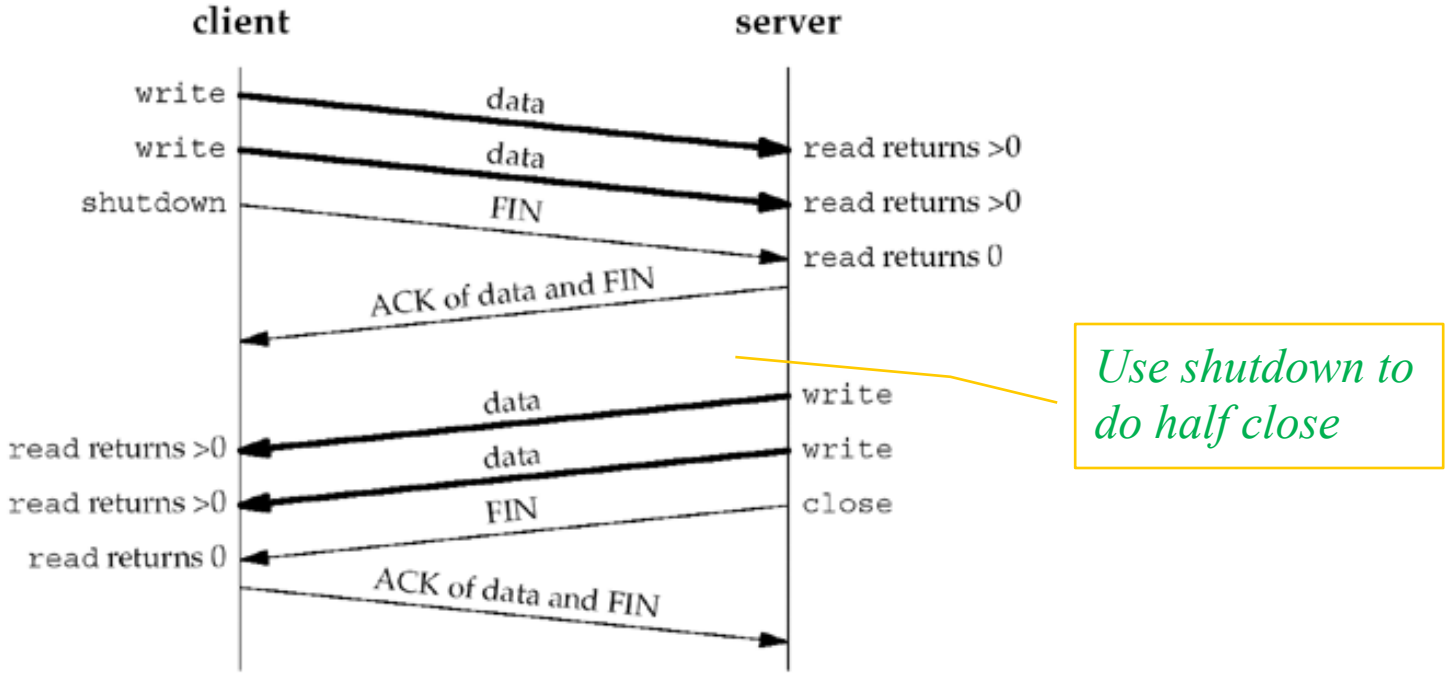
- Example: netcalc iterative server with `poll`
  - `netcalc_srv_v4`

Bulk data transfer

- TCP data transfer modes
  - Interactive (stop-and-wait)
  - Bulk
    - To transport a large amount of data, the batch mode is much more efficient
    - Flow control by the TCP sliding window
    - On the socket, it is batch input
- Example: netcalc client with batch capability
  - `netcalc_clnt_v3`

shutdown (I)

- shutdown
  - Function: shut down part of a full-duplex connection
    - Close terminates both directions of data transfer



shutdown (II)


```
int shutdown(int sockfd, int howto)
```

- `sockfd`: the socket file descriptor to be shutdown
- `howto`: the direction to shutdown
  - `SHUT_RD`: shutdown read-half of the connection
  - `SHUT_WR`: shutdown write-half of the connection
  - `SHUT_RDWR`: shutdown both read-half and write-half of the connection
- return `0` if successful, `-1` on error

Examine Socket State

- Tool: `netstat`
- Examine the socket state
  - Server starts
  - Client starts
  - Client quits before server
  - Server quits before client

SESSION 6
ADVANCED IO

Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

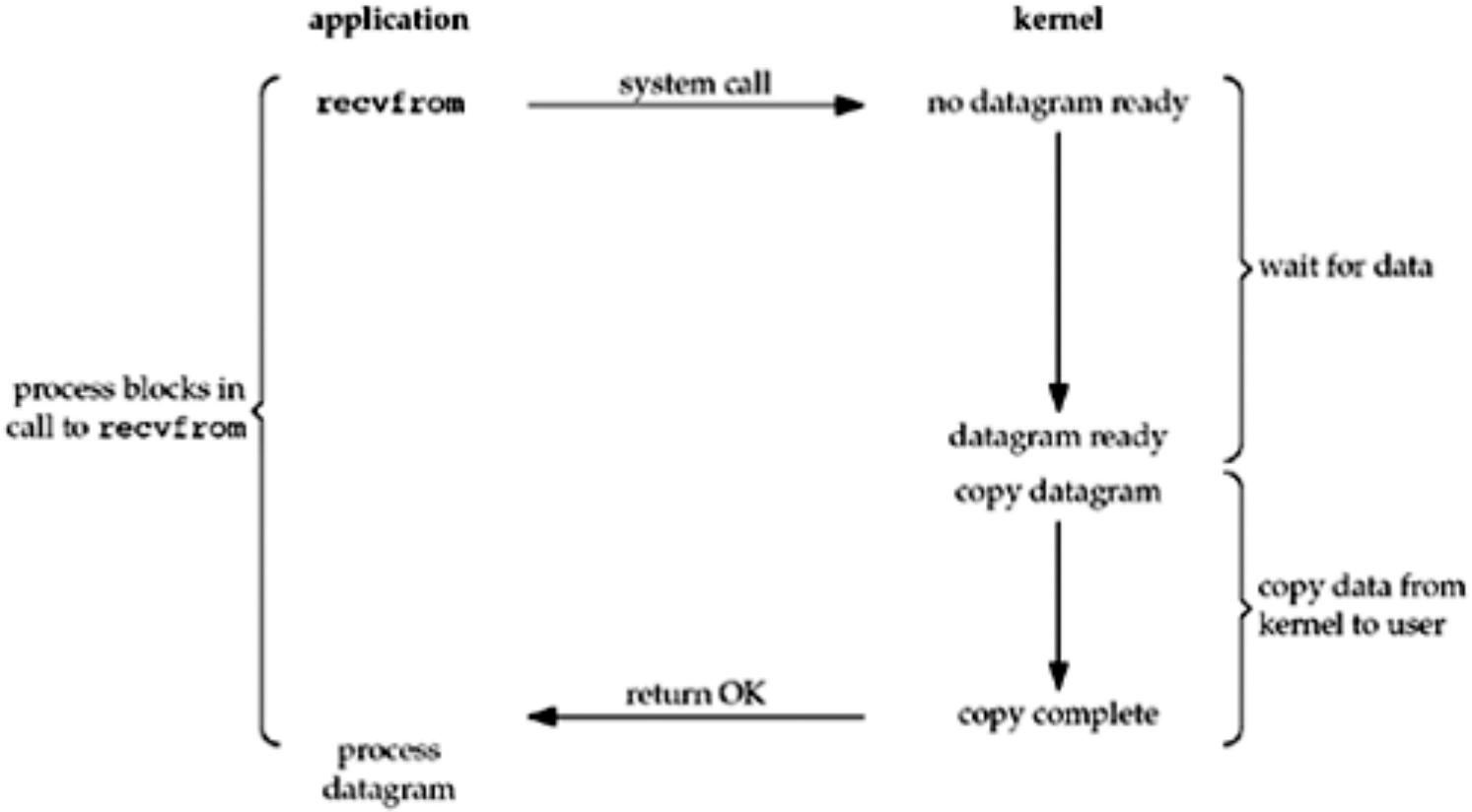
The I/O Multiplexing Requirement

- A process, either a server or a client, needs to be able to process multiple I/O events
  - A client needs to receive input from a user on the console and the response from the server on the socket
  - A server needs to accept new connection request while handling I/O on an established connection
  - A server needs to support both TCP and UDP
  - A UDP client needs to wait for the response from the server with an option to timeout

Review of UNIX/Linux I/O models

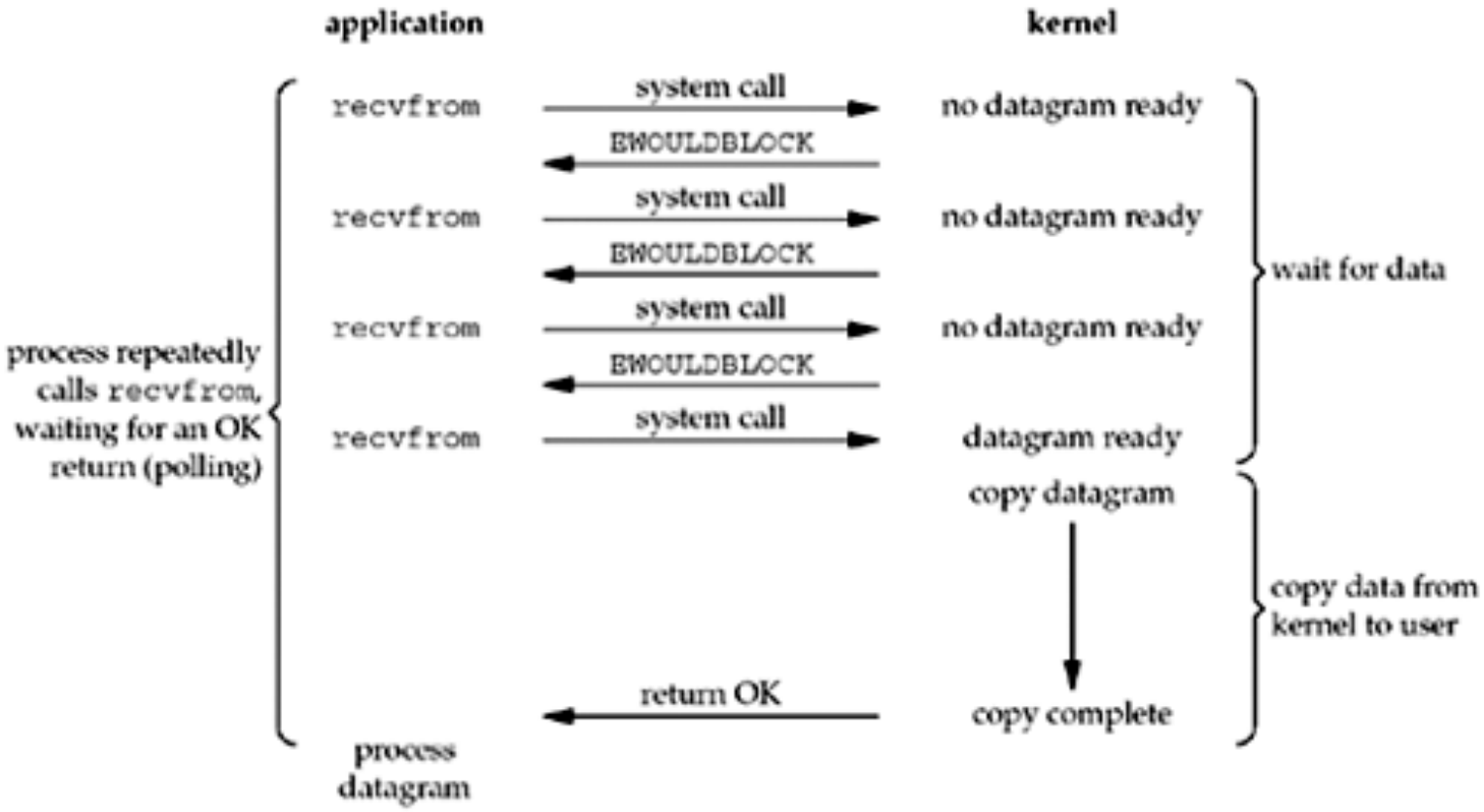
- Two phases in input operation
  - Waiting for the data to be ready
  - Copying the data from the kernel to the process
- I/O Models
  - Blocking I/O
  - Nonblocking I/O
  - I/O Multiplexing
  - Signal driven I/O
  - Asynchronous I/O

Blocking I/O models



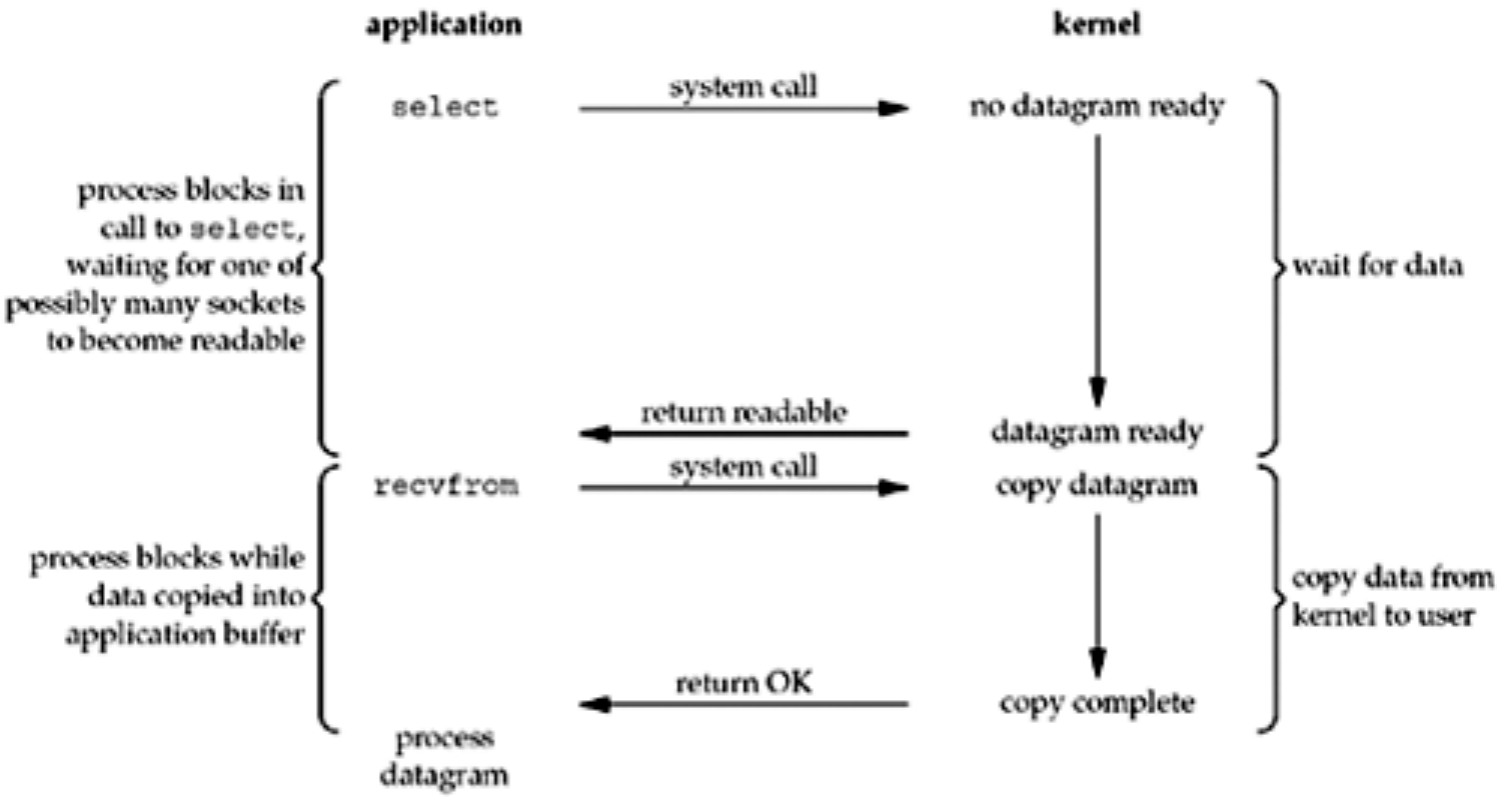
The process is in the Blocked state, and can not do anything else

Nonblocking I/O models



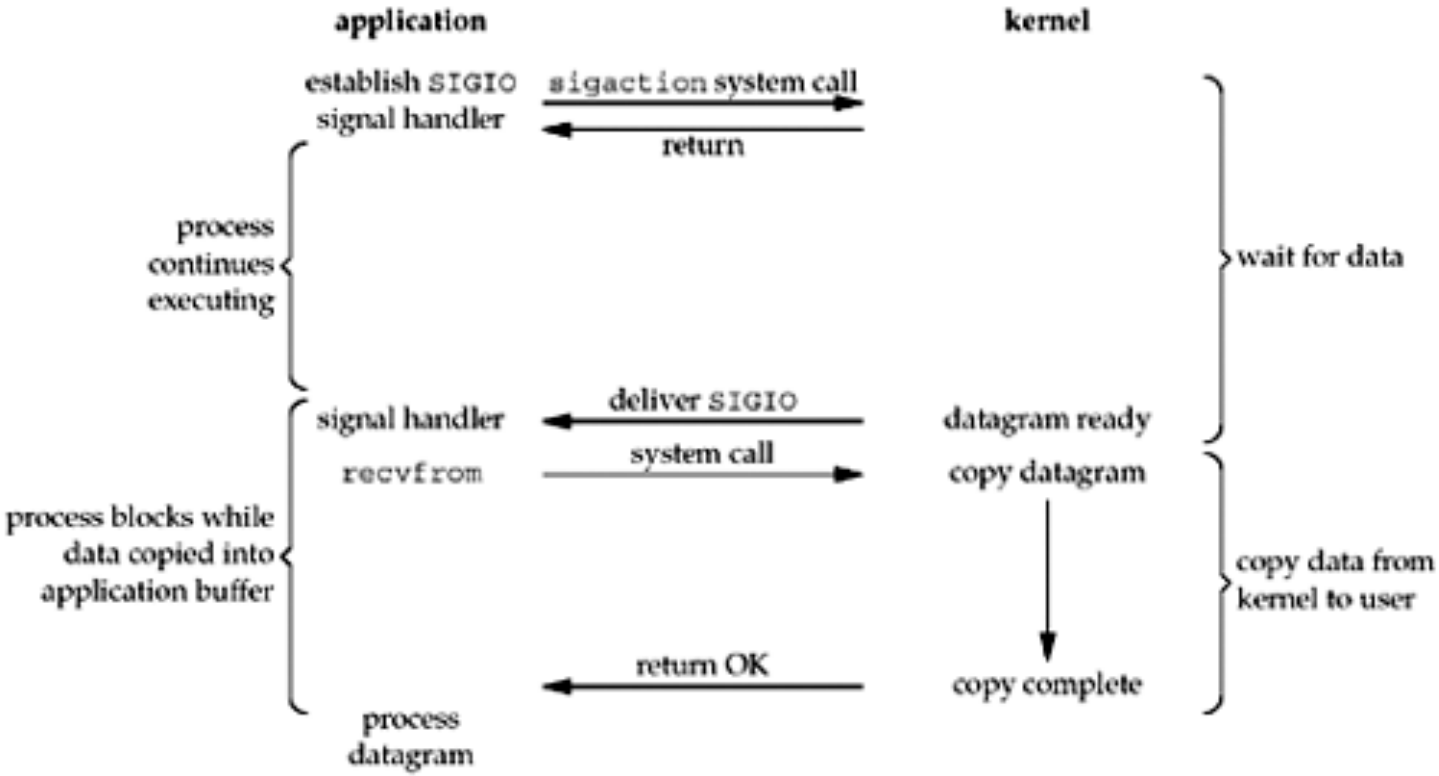
The process is not stuck in the Blocked state, so it can do something else if required

I/O Multiplexing



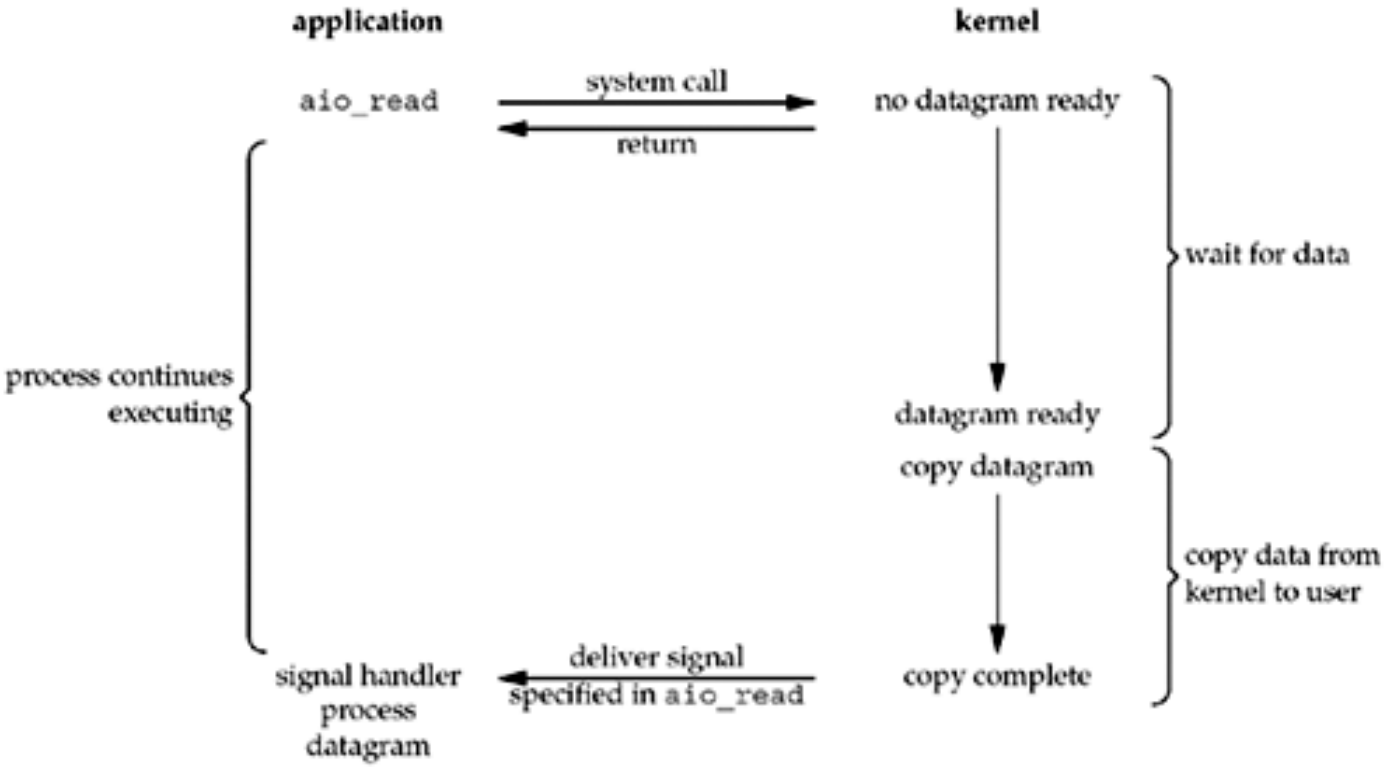
The process is blocked on multiple I/O sources (basically moving the polling down to the kernel)

Signal Driven I/O Model



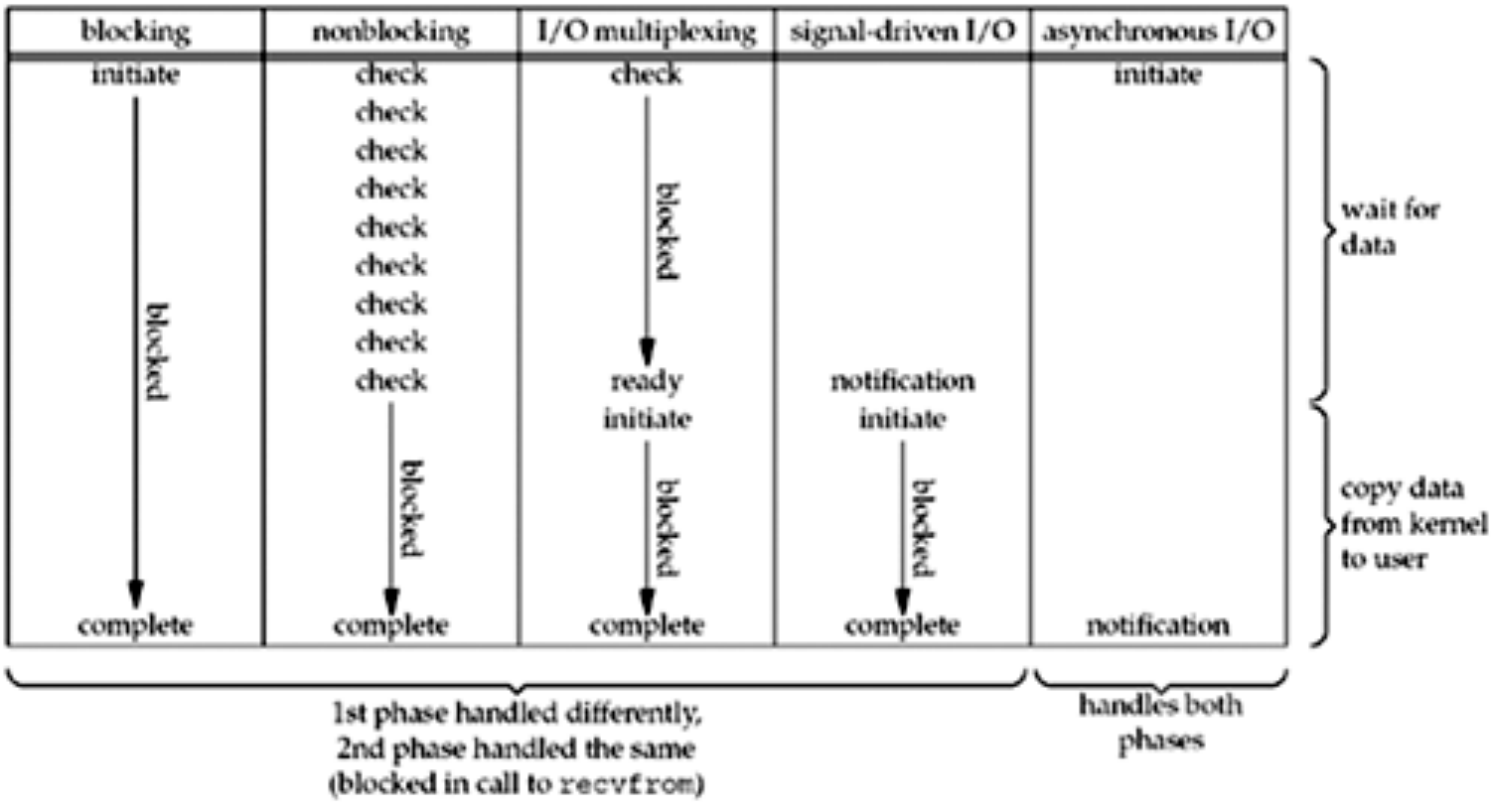
The process is blocked until it receives an asynchronous notification from the kernel

Asynchronous I/O Model



The process is immediately unblocked, and the signal is used to notify the I/O completion. May not work on some servers because it is only supported on the real time version of the Linux. Rarely used.

Comparisons of the I/O Models



Examples of the I/O Models

I/O Models	Examples	Comments
Blocking		Most of the examples are blocking
Nonblocking	testprog/async_io.c unpv13e/nonblock/ daytimetcpcli.c	the file IO example is better in illustrating the nonblocking feature
I/O multiplexing	netcalc_srv_v4 netcalc_clnt_v3	The program that uses either poll or select
Signal Driven	unpv13e/advio/tcpcli01 unpv13e/advio/udpcli03	udpcli03 uses dg_cli from dgclitimeo3.c; use server udpcliserv/udpservselect01 to support both clients
Asynchronous IO		Not that popular. See more info http://fwheel.net/aio.html or google "AIO Linux"

Blocking Calls in Socket

- By default, sockets are blocking
  - Input blocking (read, recv, recvfrom etc.)
  - Output blocking (write, send, sendto etc.)
  - Accepting incoming connections (accept)
  - Initiating outgoing connections (connect)
- The various none blocking I/O methods can be applied to all four types of the blocking in the sockets
  - The built in support in the flag `MSG_DONTWAIT` in some of the socket IO APIs

Server Design Choice

- Server design
  - Iterative server
    - Multiplex on multiple socket file descriptors
  - Concurrent server
    - Spawn a child server to service a client
- Example: netcalc iterative server
  - `netcalc_srv_v4`

SESSION 7
UDP SOCKETS

Bin Zhang ◆ NPU (binzhang@mail.npu.edu)

Overview of UDP Client-server

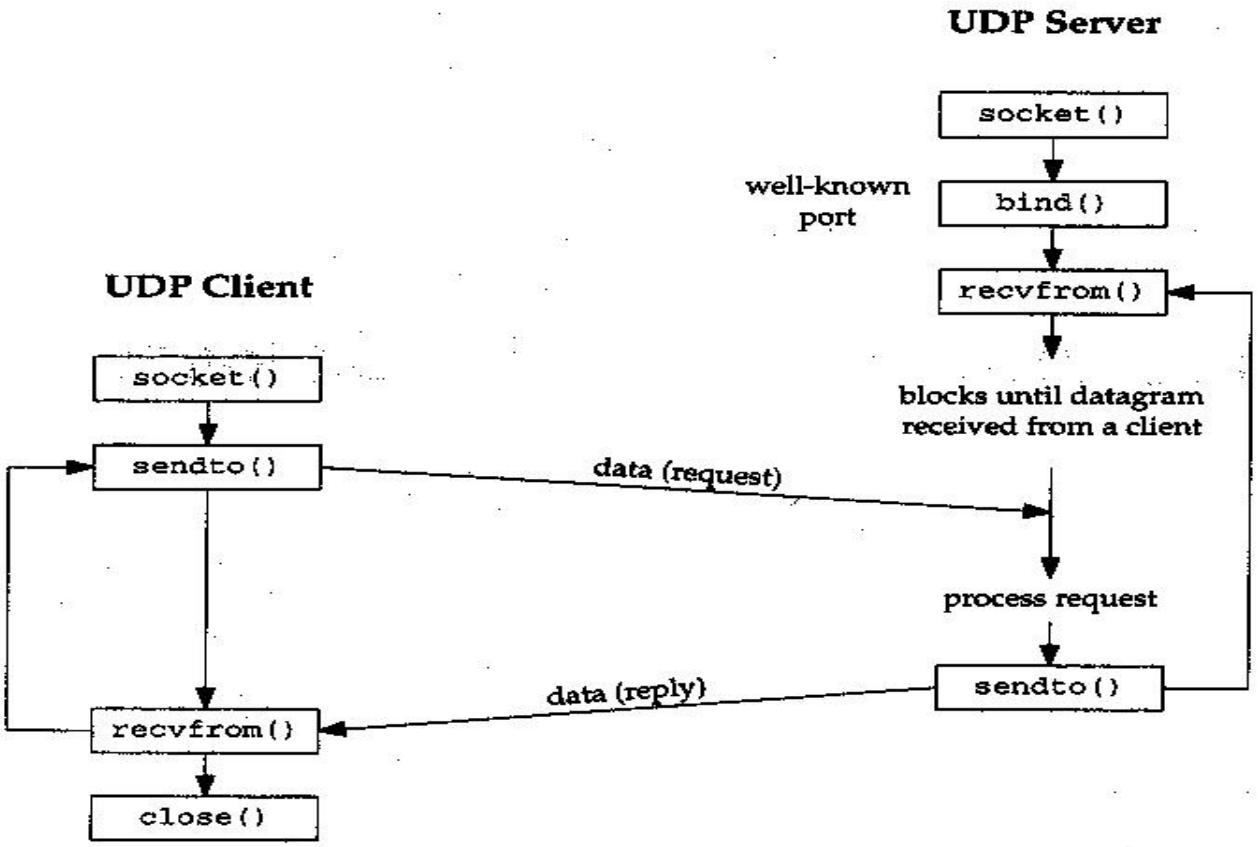


Figure 8.1 Socket functions for UDP client–server.

sendto (I)

- **sendto**
  - Function: send a message through a socket to a peer

```
ssize_t sendto(int sockfd, const void *buff,
size_t nbytes, int flags, const struct sockaddr
*to, socklen_t addrlen)
```

    - **sockfd**: the socket file descriptor to send message through
    - **buff**: the pointer to a buffer containing the message to be sent
    - **nbytes**: the length of the message in bytes
    - **flags**: the types of message transmission
    - **to**: the pointer to a **sockaddr** structure containing the destination address

sendto (II)

- `addrlen`: the length of the `sockaddr` structure pointed by the `to`
- return the number of bytes sent if successful, `-1` otherwise

recvfrom (I)

- **`recvfrom`**
  - Function: receive a message from a socket

```
ssize_t recvfrom(int sockfd, void *buff, size_t
nbytes, int flags, struct sockaddr *from,
socklen_t *addrlen)
```

    - **`sockfd`**: the socket file descriptor to receive message from
    - **`buff`**: the buffer pointer where the message should be stored
    - **`nbytes`**: the length in bytes of the buffer
    - **`flags`**: the type of message reception (use `0` for this class)
    - **`from`**: the pointer to a **`sockaddr`** structure in which the sending address is stored

MSG_DONTWAIT is a non-blocking flag

recvfrom (II)

- **`addrlen`**: the length of the **`sockaddr`** structure pointed by the **`from`**
  - This is a value-result argument
- return the length of the message if successful, `-1` otherwise

`recvfrom` may block forever if no data comes at all

- The solution is for the application layer to recover
  - Use a timer
  - I/O multiplexing

UPD Server

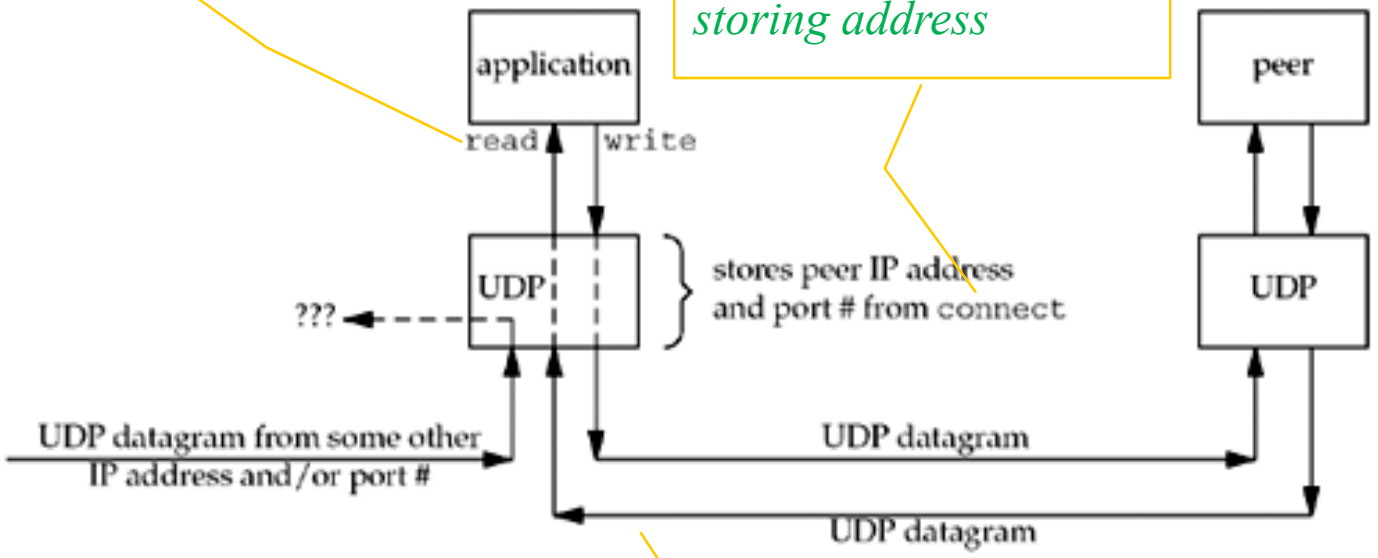
- Most of the UDP servers are iterative
  - Most of the TCP servers are concurrent
- Example: netcalc with UDP
  - `netcalc_clnt_udp`
  - `netcalc_srv_v6`
- Problems for using UDP sockets
  - Lost datagram
  - Lack of flow control
  - It is the responsibility of the application layer to solve these problems
    - Usually live with them to some extent; otherwise, why not using TCP?

Connected UDP Socket

Why connected?
- *Security*
- *Return send error*

Do not use sendto and recvfrom

Call connect, but does not result in TCP-style connection. Only storing address



Either client or server or both can set up connected UDP socket