UNIX Networking

Peter Sjödin

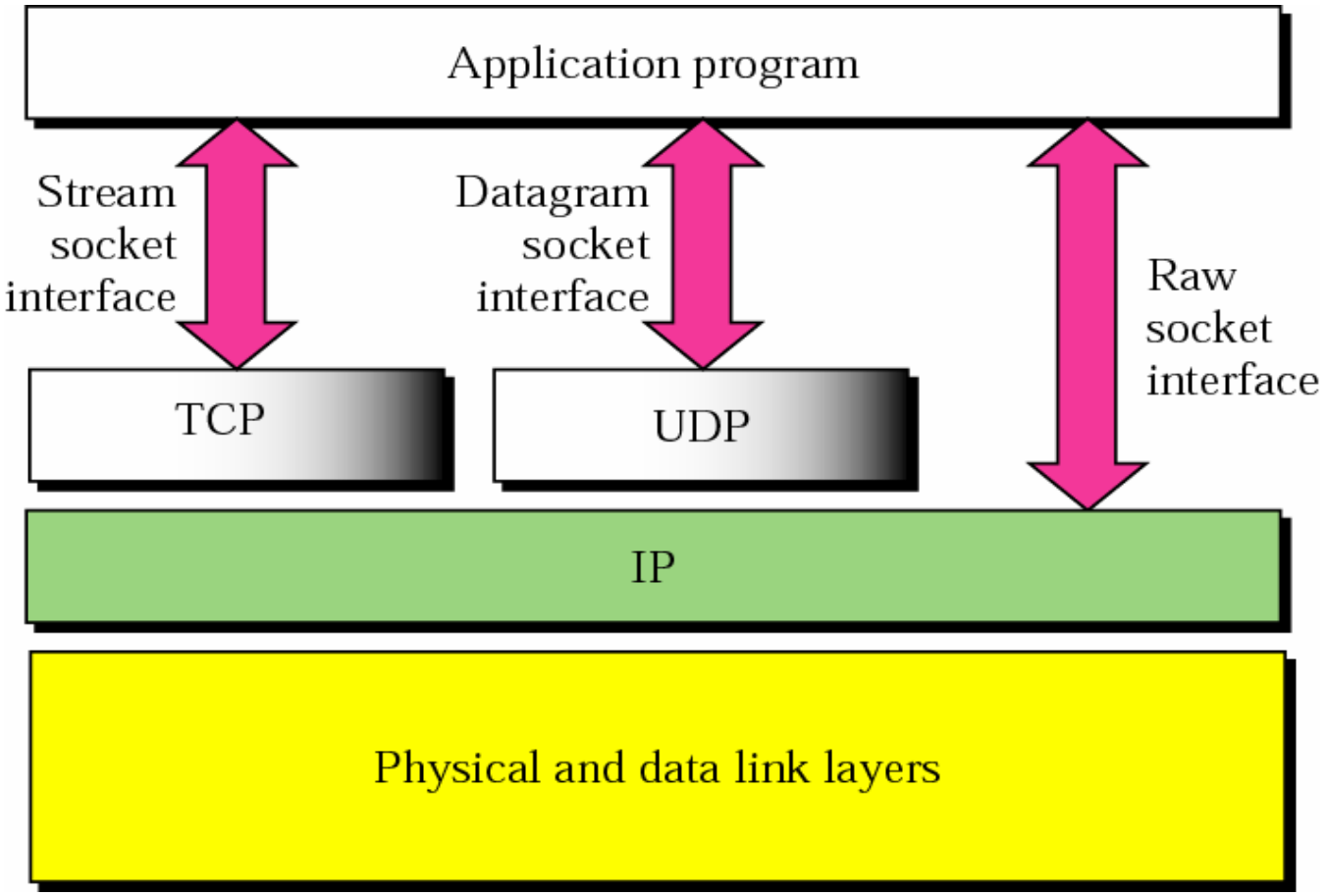
Outline

- Socket programming in C
- I/O Models
- Kernel programming
- Packet buffers

Sockets

- Abstraction that applications can use for sending and receiving data
  - In analogy with files for reading and writing data on external storage
- Berkeley Sockets Interface API
  - "Sockets" for short
- Originally developed for Berkeley UNIX (BSD 4.1)
  - Early 80's
  - Now used for all UNIX versions
  - Windows (WinSock API), Java (Socket, ServerSocket objects), ...
- Typical use is for a transport service (TCP or UDP)

Socket Types



From B. A. Forouzan: Data Communications and Networking, 3rd ed, McGraw-Hill

Creating Sockets

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

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

- There are two common variants

```
s = socket(PF_INET, SOCK_STREAM, 0);
```

```
s = socket(PF_INET, SOCK_DGRAM, 0);
```

- Protocol zero means default protocol for the socket type (and domain)
  - TCP for PF_INET + SOCK_STREAM
  - UDP for PF_INET + SOCK_DGRAM

Sockets Are for Networks in General…

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

Domain (family)	*Purpose*	*Man page*
PF_UNIX, PF_LOCAL	Local communication	unix(7)
PF_INET	IPv4 Internet protocols	ip(7)
PF_INET6	IPv6 Internet protocols	
PF_IPX	IPX - Novell protocols	
PF_NETLINK	Kernel user interface device	netlink(7)
PF_X25	ITU-T X.25 / ISO-8208 protocol	x25(7)
PF_AX25	Amateur radio AX.25 protocol	
PF_ATMPVC	Access to raw ATM PVCs	
PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Low level packet interface	packet(7)

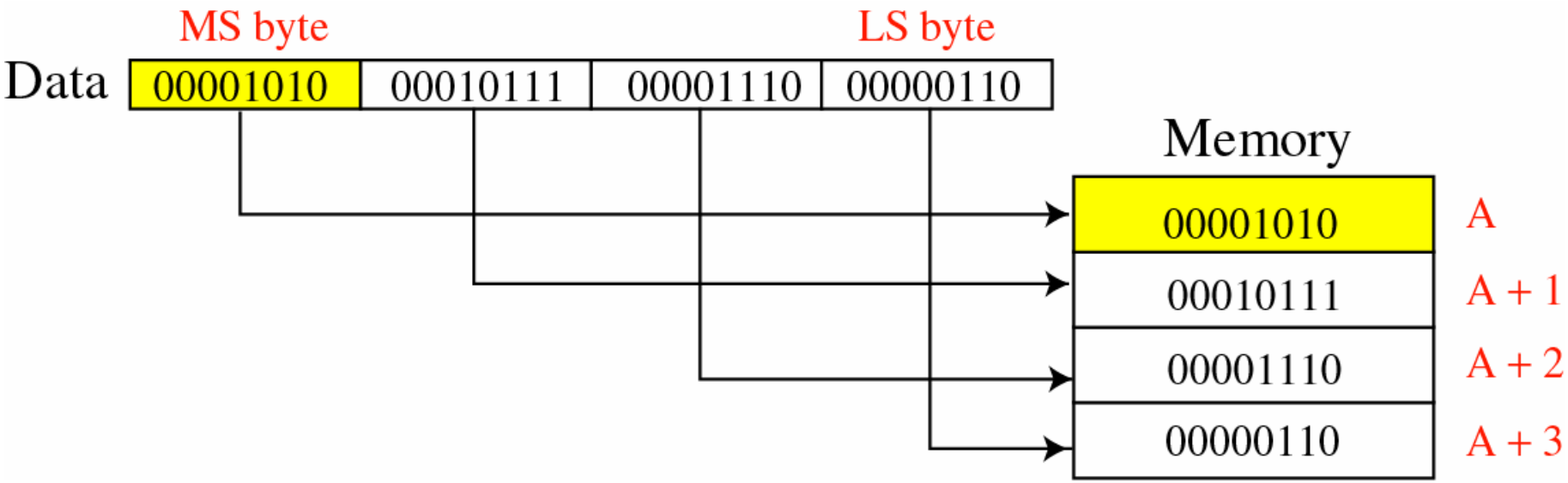
Type	*Purpose*
SOCK_STREAM	Sequenced, reliable, two-way, connection-based byte streams.
SOCK_DGRAM	Datagrams.
SOCK_SEQPACKET	Sequenced, reliable, two-way connection-based data transmission path for datagrams
SOCK_RAW	Raw network protocol access.
SOCK_RDM	A reliable datagram layer that does not guarantee ordering.
SOCK_PACKET	Obsolete and should not be used in new programs

Protocol	*Purpose*
IPPROTO_TCP	TCP (Only legal protocol for PF_INET + SOCK_STREAM)
IPPROTO_UDP	UDP (Only legal protocol for PF_INET + SOCK_DGRAM)
0	Default protocol for given domain and type

Socket Addresses

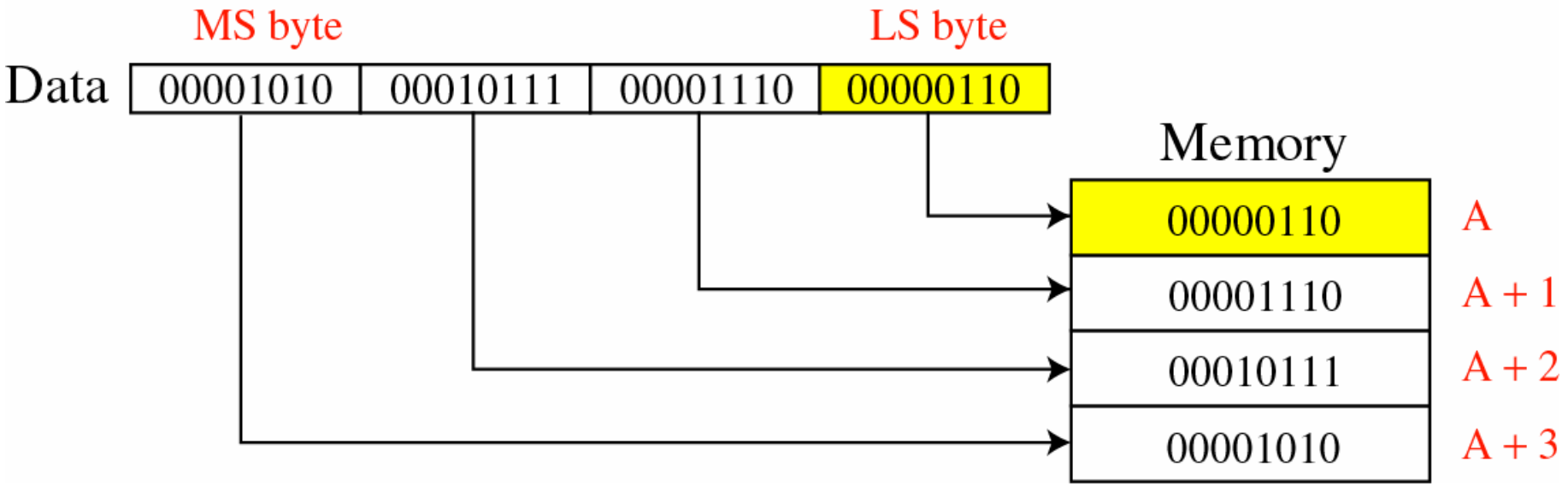
- Need to be able to specify addresses
- Clients needs to
  - Specify the IP address of the destination
  - Specify the port number of the destination
  - (Obtain a local port number)
- Servers need to
  - Associate the (well-known) server port number to the socket
  - (Restrict the service to particular IP addresses)
- Addresses are specified in network byte order (which is big endian)

Big-Endian Byte Order



From B. A. Forouzan: TCP/IP Protocol Suite, 2nd ed, McGraw-Hill

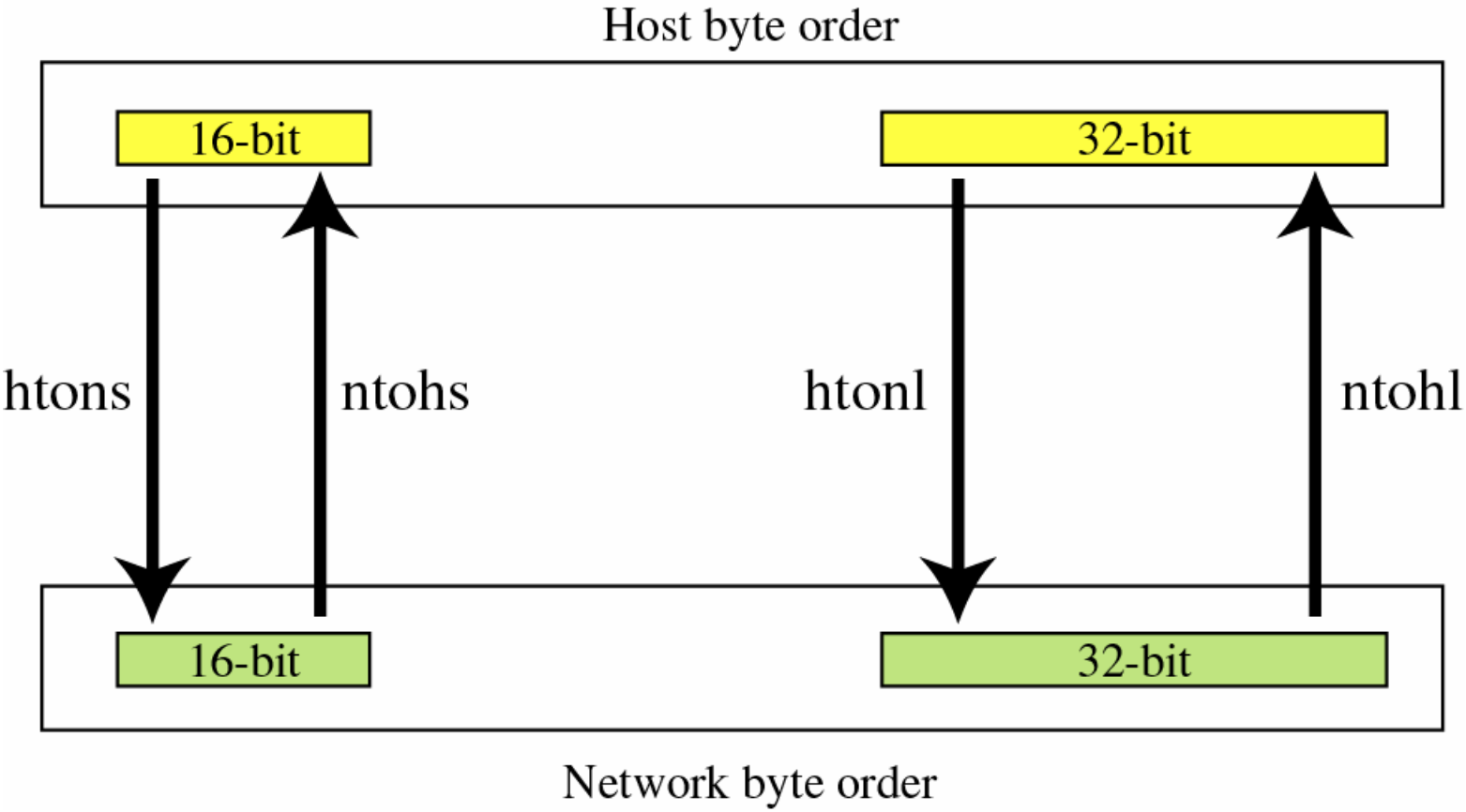
Little-Endian Byte Order



From B. A. Forouzan: TCP/IP Protocol Suite, 2nd ed, McGraw-Hill

Byte-Order Transformation

- "short"—16 bits
- "long"—32 bits



From B. A. Forouzan: TCP/IP Protocol Suite, 2nd ed, McGraw-Hill

IP Socket Addresses

```
struct sockaddr_in {
    sa_family_t      sin_family;    /* address family: AF_INET */
    u_int16_t        sin_port;      /* port in network byte order */
    struct in_addr   sin_addr;      /* internet address */
};

/* Internet address. */
struct in_addr {
    u_int32_t        s_addr;        /* address in network byte order */
};
```

- Small variations between different operating systems
- `inet_addr()` can be used to convert a dotted-quad string to a binary representation
  - in network byte order

```
in_addr_t inet_addr(const char *cp);
```

Using DNS

```
struct hostent *gethostbyname(const char *name);

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

- Typically just grab the first address

```
char *name;
struct hostent *host;
struct in_addr result;

if ((host = gethostbyname(name)) == NULL)
    ExitWithError("Can't lookup host");
if (host->h_addrtype == AF_INET)
    result.s_addr = *((u_int32_t *) host->h_addr_list[0]);
else
    ExitWithError("Not an IPv4 address\n");
```

Opening a Connection

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

- Connect the socket
  - Establish TCP connection (SOCK_STREAM)
  - Can also be used for UDP sockets (SOCK_DGRAM), but that is less common
    - Specify remote peer for subsequent datagrams

Connect Example

```
int s;
int ServerPort;
char *ServerAddress;
struct sockaddr_in ServerAddr;

if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    ExitWithError("socket failed");
}
memset(&ServerAddr, 0, sizeof(ServerAddr); /* Clear */
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_addr.s_addr = inet_addr(ServerAddress);
ServerAddr.sin_port = htons(ServerPort);

if (connect(s, (struct sockaddr *) &ServerAddr, sizeof(ServerAddr)) < 0)
{
    ExitWithError("connect failed");
}
```

Specifying Server Address

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

- Assign an address to the socket
- Bind deals with the **local address**
  - Connect deals with the **remote address**

Receiving Data

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sock- addr *from, socklen_t *fromlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

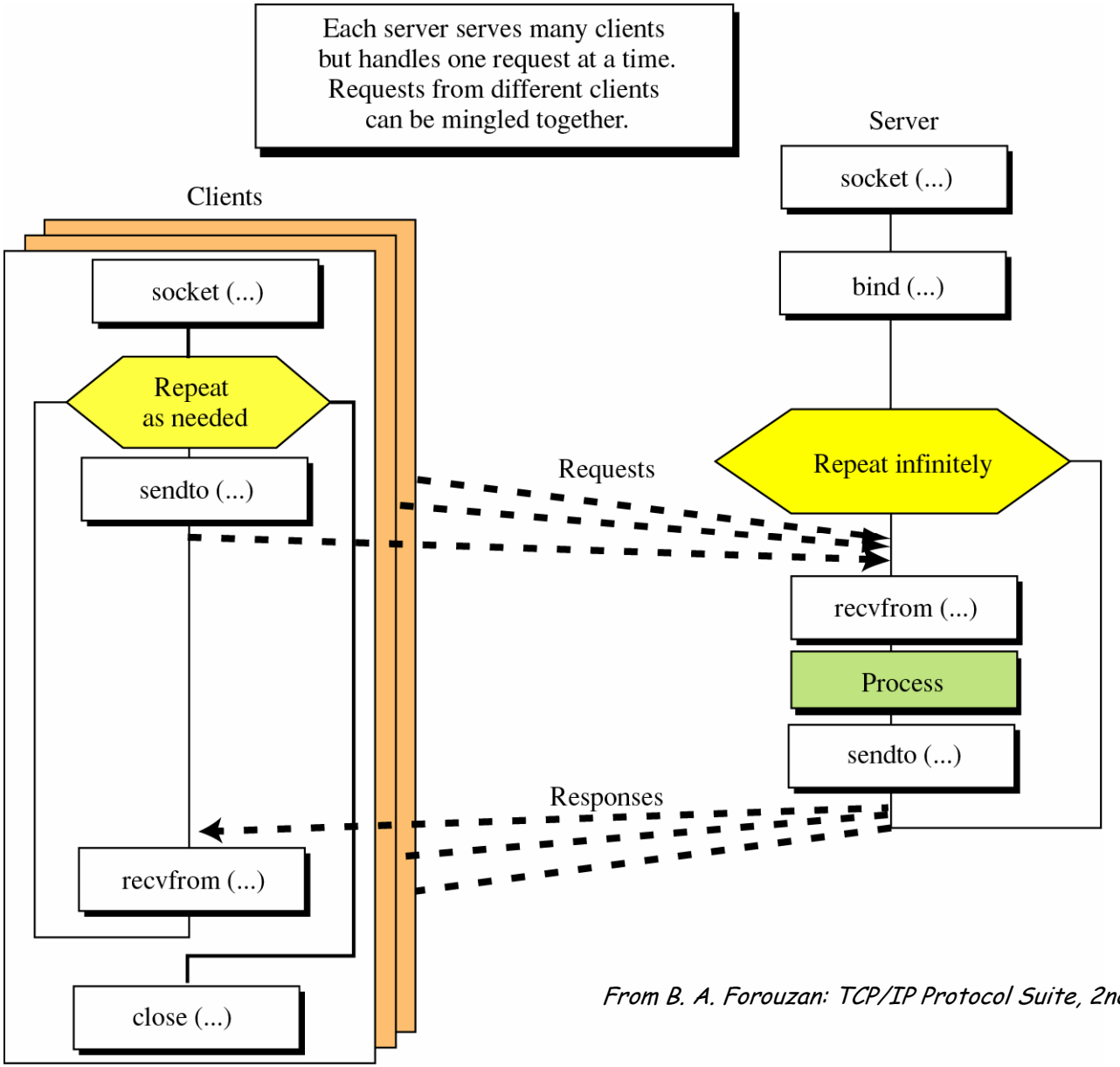
- `Recv` and `read` calls are normally only used with connected sockets
- `Recvfrom` and `recvmsg` may be used whether or not the socket is connected
  - The source address of the message is filled in

Sending Data

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t send(int s, const void *msg, size_t len, int flags);
ssize_t sendto(int s, const void *msg, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

- `Send` and `write` calls are normally only used with connected sockets
- `Sendto` and `sendmsg` may be used whether or not the socket is connected
  - The destination address of the message is given

Connectionless Server



From B. A. Forouzan: TCP/IP Protocol Suite, 2nd ed, McGraw-Hill

Allowing Incoming Connections

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

- TCP server side
- The `backlog` parameter defines the maximum length the queue of pending connections may grow to
  - Queue length for completely established sockets waiting to be accepted
    - As of Linux 2.2
  - Used to be the number of incomplete connection requests
  - TCP SYN DoS attacks

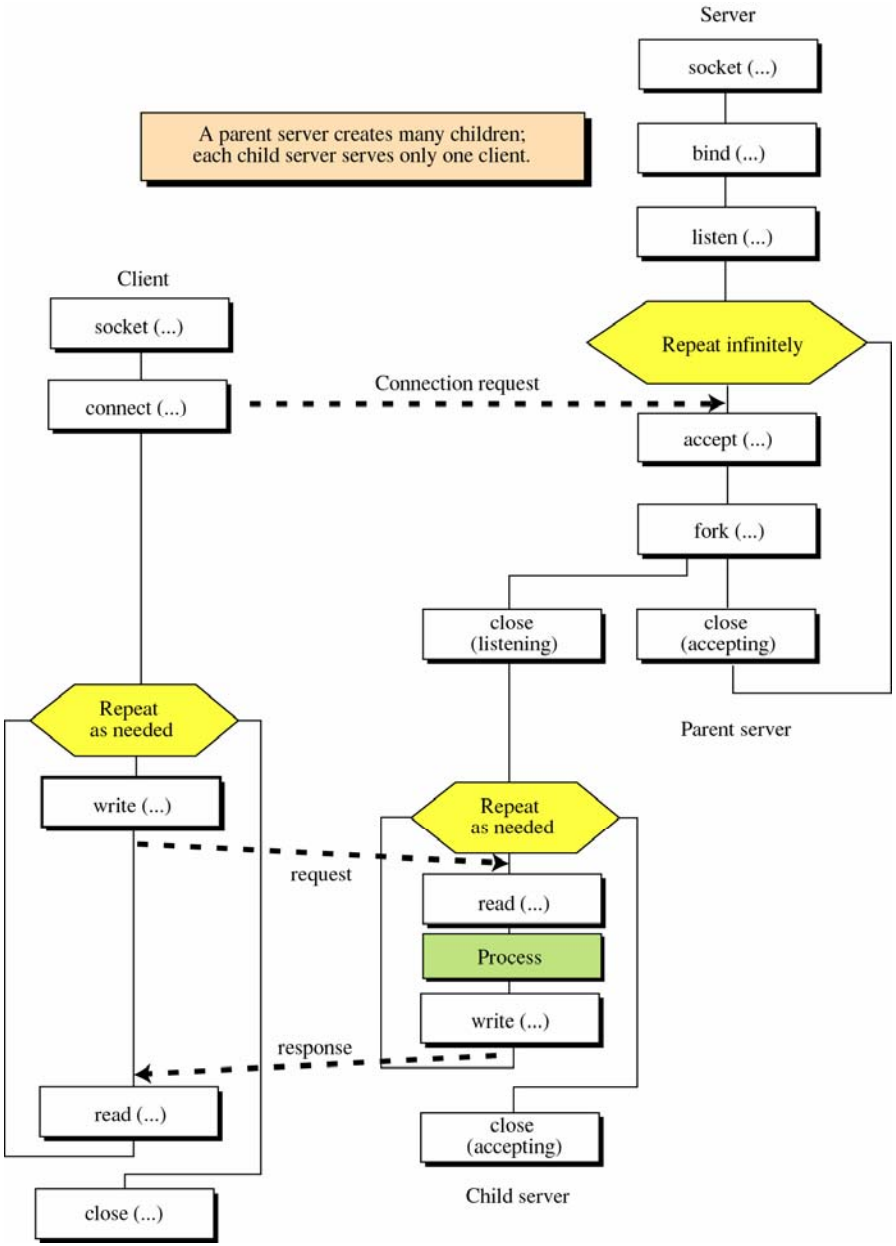
Get Next Incoming Connection

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

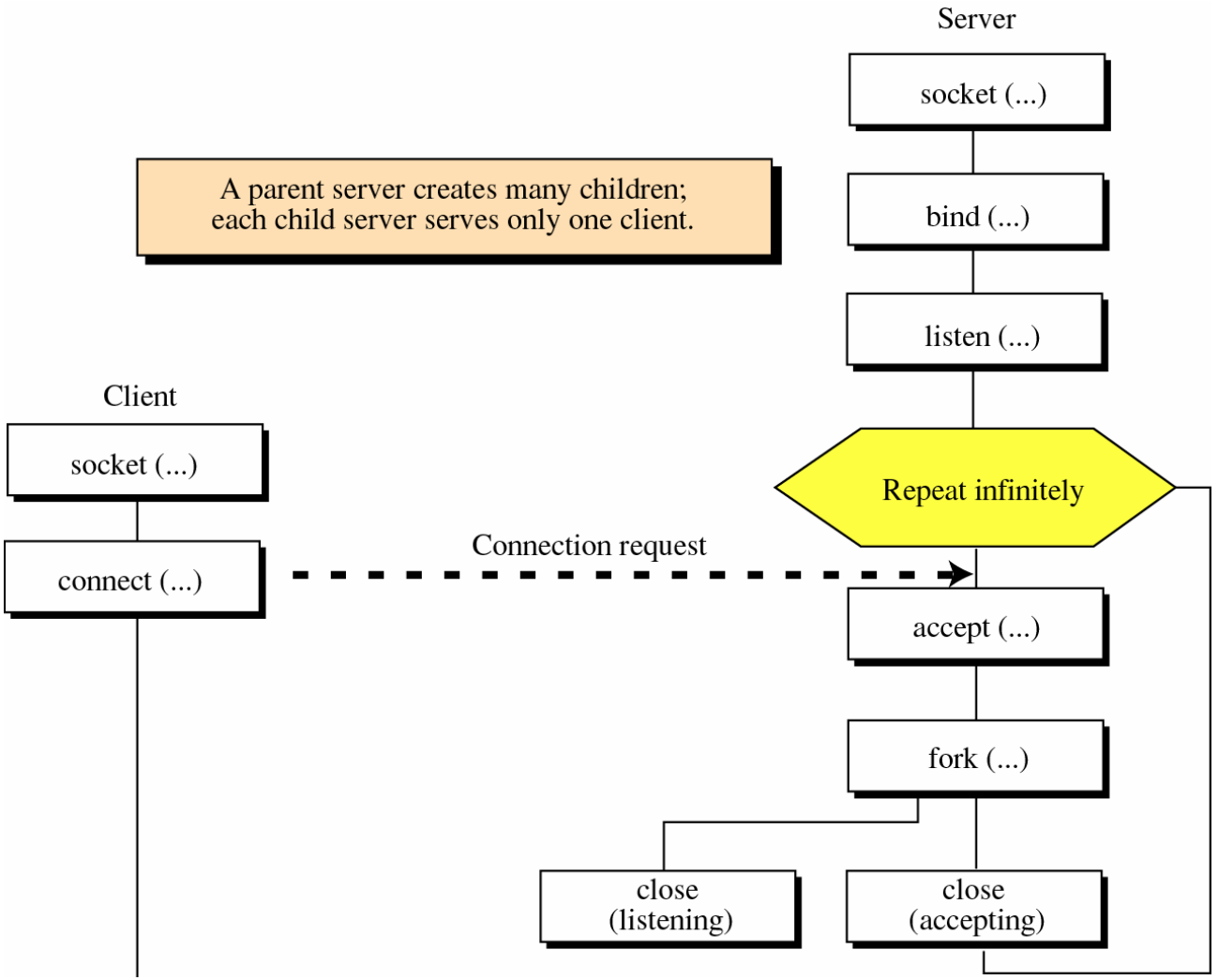
- TCP server side
- Get next pending connection from queue
- Return a new socket descriptor, that can be used for communication with the client
  - New socket is connected, but not in listening state
- Need to keep the original socket, for accepting more connections

Connection-Oriented Server

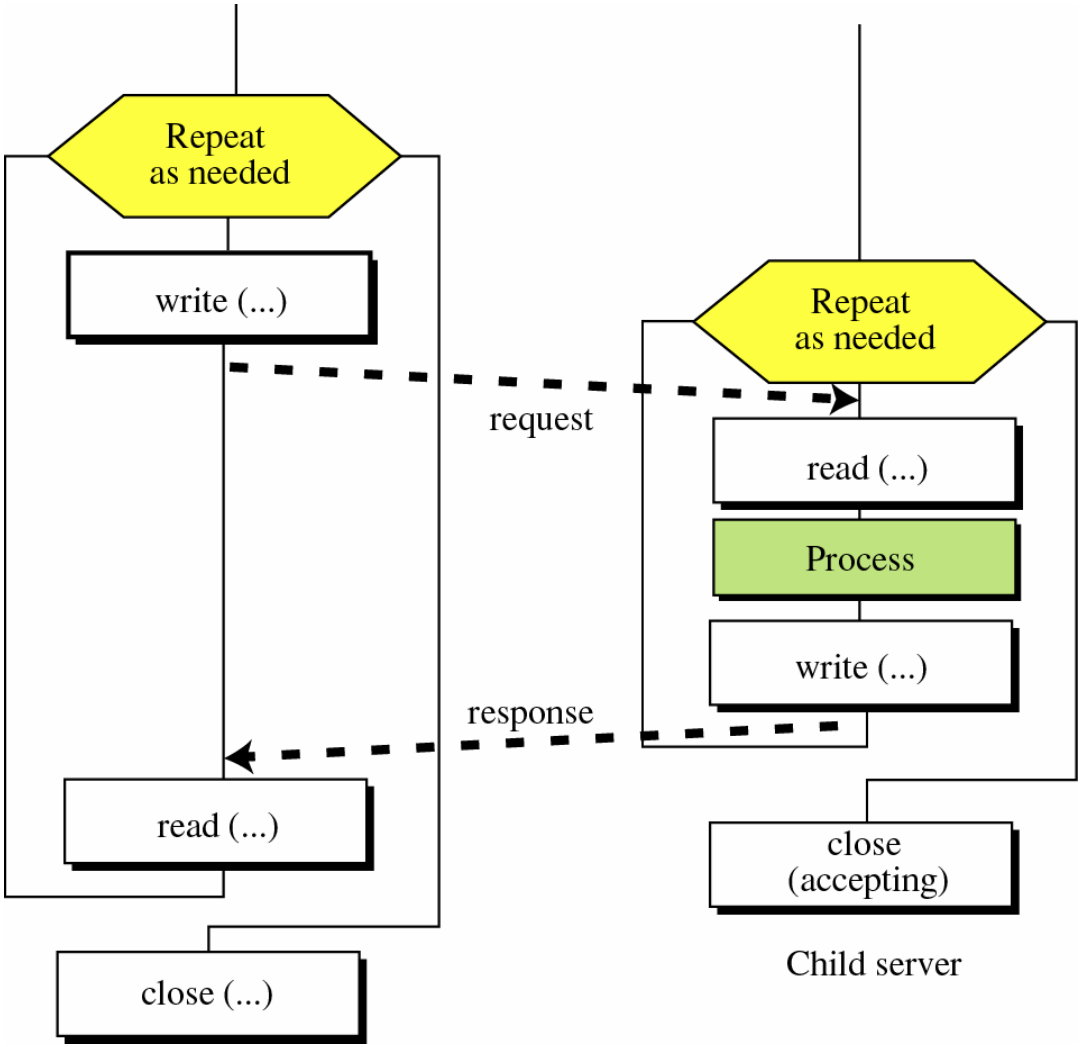
- Server has two parts
  - One that processes incoming connections
  - One that serves clients
- Multiplexing
  - Associate client state with file descriptor
- Multiple processes
  - Often one process per client
    - Fork (heavyweight)
    - Threads (lightweight)



Connection Establishment Phase



Data Transfer Phase



Requesting Socket Addresses

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

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

- Getpeername
  - Get name (socket address) of the peer connected to the socket
  - Remote side
- Getsockname
  - Get name (socket address) of the socket
  - Local side

Socket Options

```
int getpeername(int s, struct sockaddr *name, socklen_t
*namelen);
int getsockname(int s, struct sockaddr *name, socklen_t
*namelen);
```

- Attributes related to socket behavior
- Exist at different protocol levels
  - SOL_SOCKET, SOL_IP, SOL_TCP

Socket Options

SOL_SOCKET	***Purpose***
SO_SNDBUF	**Maximum send buffer size**
SO_RCVBUF	**Maximum receive buffer size**
SO_KEEPALIVE	**Enable/disable keep-alive messages (connection-oriented sockets)**
SO_BROADCAST	**Allow sending/reception of broadcast packets (datagram sockets)**
SO_REUSEADDR	**Allow reuse of local address in bind calls**

SOL_IP	***Purpose***
IP_TTL	**TTL field for packets sent**
IP_ADD_MEMBERSHIP	**Join a multicast group**
IP_DROP_MEMBERSHIP	**Leave a multicast**
IP_MTU	**Get current path MTU (connected sockets)**

SOL_TCP	***Purpose***
TCP_MAXSEG	**Maximum segment size**
TCP_NODELAY	**Disable the Nagle algorithm**

- See "man 7 socket", "man 7 ip", "man 7 tcp", ...

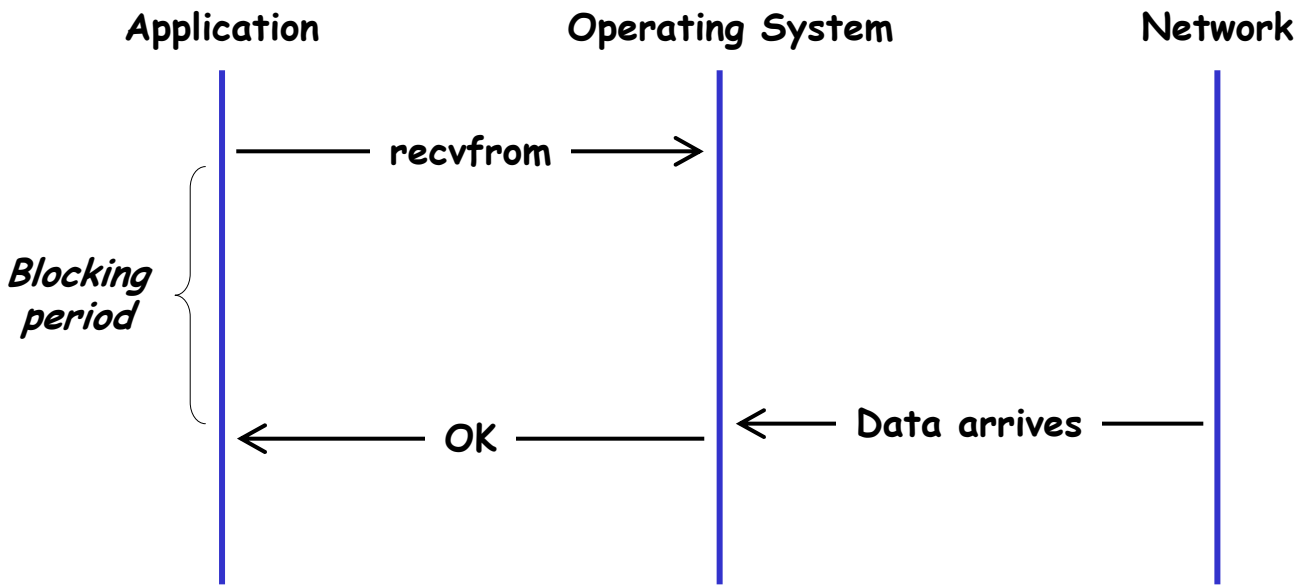
Summary

- Create socket
  - **Socket**
- Client side
  - **Connect**
- Server side
  - **Bind**
  - **Listen, accept**
- Data transfer
  - **Write, send, sendto, sendmsg**
  - **Read, recv, recvfrom, recvmsg**
- Socket information and attributes
  - **Getsockname, getpeername**
  - **Getsockopt, setsockopt**

I/O Models

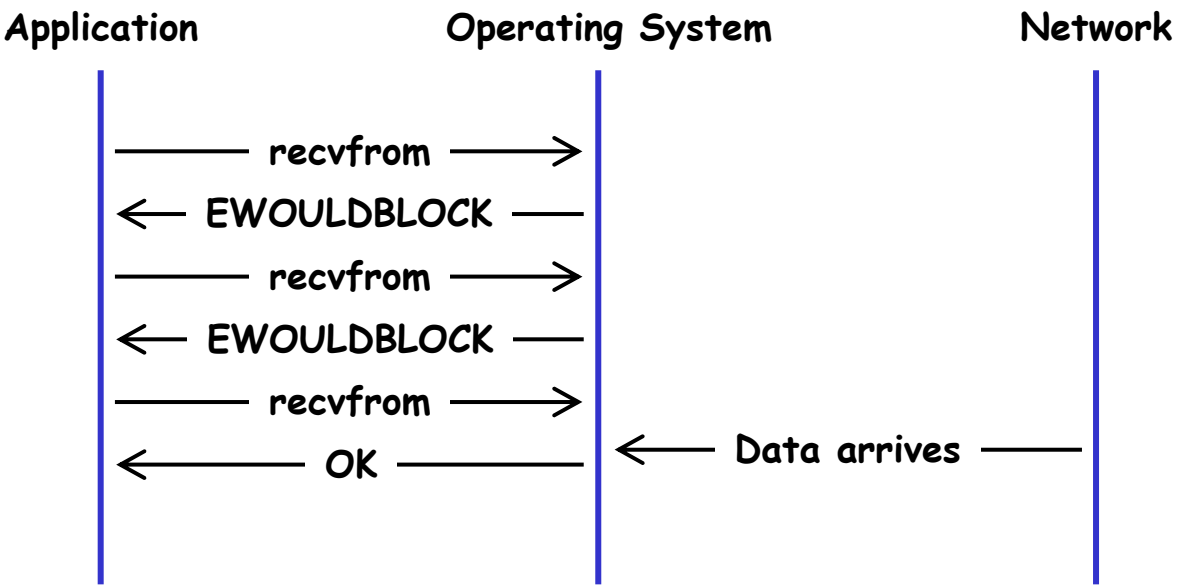
- How do applications communicate with kernel networking layer?
- In particular, how do applications receive data?
- Different models
  - Blocking
  - Non-blocking
  - Asynchronous

Blocking I/O



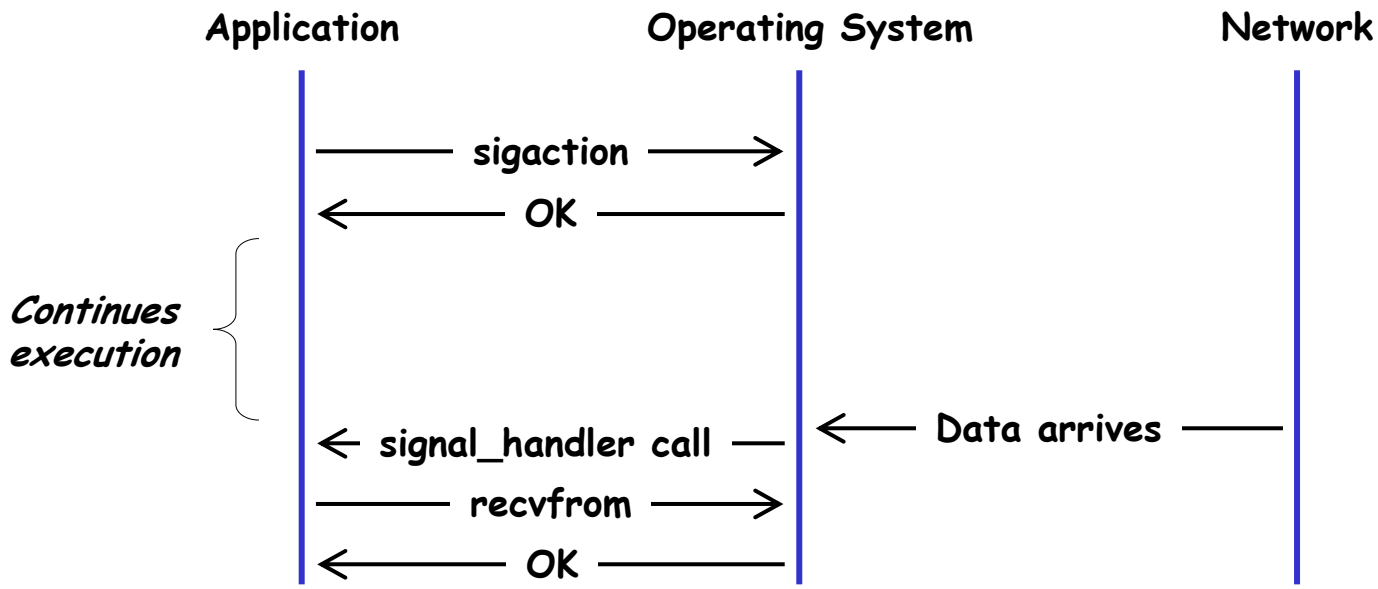
- Caller is blocked (suspended) until data is available
- Sockets are normally blocking
  - Read, recv, recfrom, etc

Non-blocking I/O



- Caller continuously checks until data is available
- Sockets can be configured as non-blocking
- Fcntl system call
  - O_NONBLOCK flag with F_SETFL command
- ioctl system call
  - FIONBIO command

Asynchronous I/O

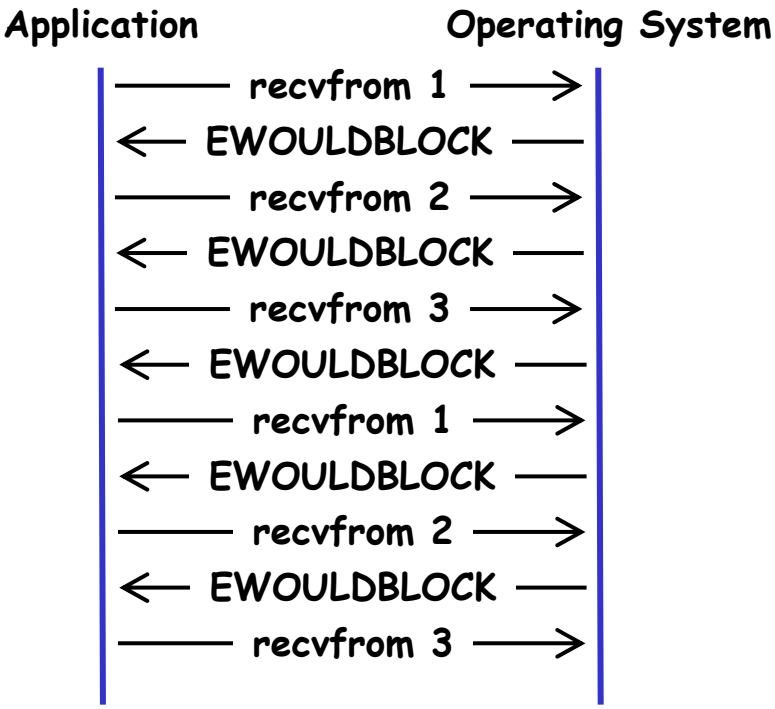


- Signal-based
- Register SIGIO signal handler
  - sigaction system call
- Signal handler gets called when data is available
  - Reads data in the normal way

Multiplexing

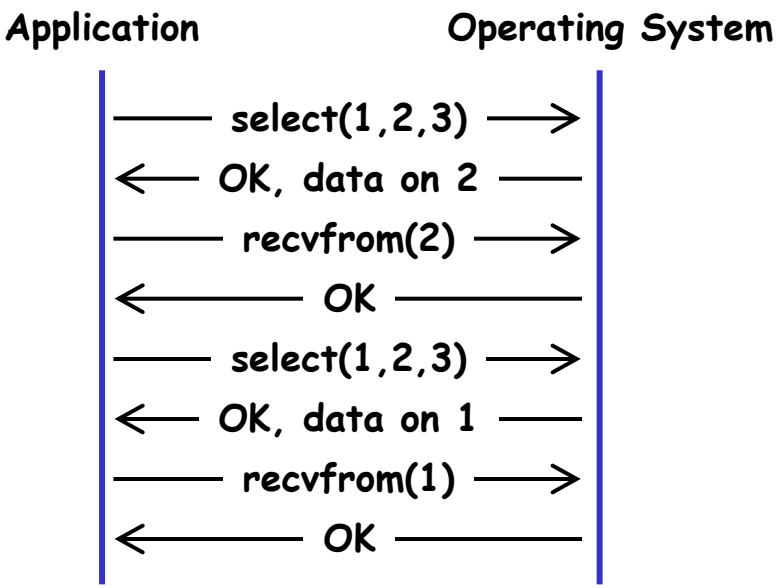
- How to deal with multiple sockets
  - Do not know in advance which descriptor will be ready
    - Cannot use regular blocking I/O
- Three methods
  1) Non-blocking I/O
     - Polling
  2) Blocking I/O on multiple descriptors
  3) Multiple threads

Polling



- Busy-wait loop
- Mostly used in dedicated systems

Blocking I/O on Multiple Descriptors



- Caller is blocked (suspended) until data is available
- Sockets are normally blocking
  - Read, recv, recfrom, etc
- Select and poll functions

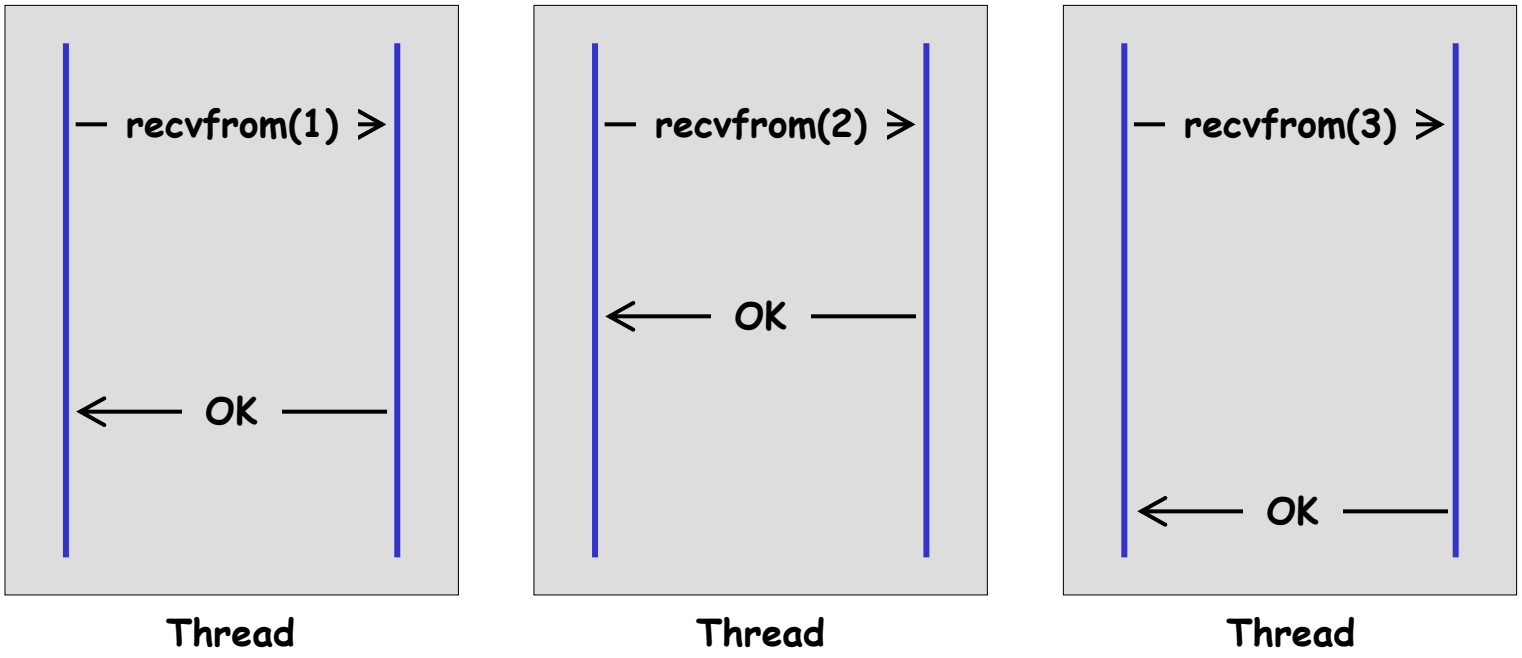
Select System Call

```
int
select(nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
       struct timeval *timeout);
```

- fd_set structure represents set of file descriptors
  - Passed as call-by-reference
    - Modified on return to reflect file descriptors with pending events
- Select is blocking
- Timeout argument to limit blocking time
  - Used for timers, polling
- Macros for accessing fd_set structures

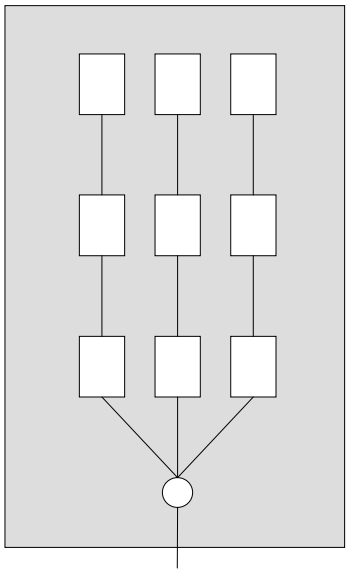
```
FD_ZERO(fd_set *)              Initialize fd_set to be empty
FD_CLR(int fd, fd_set *)       Remove fd from fd_set
FD_SET(int fd, fd_set *)       Add fd to fd_set
FD_ISSET(int fd, fd_set *)     Check if fd is present in fd_set
```

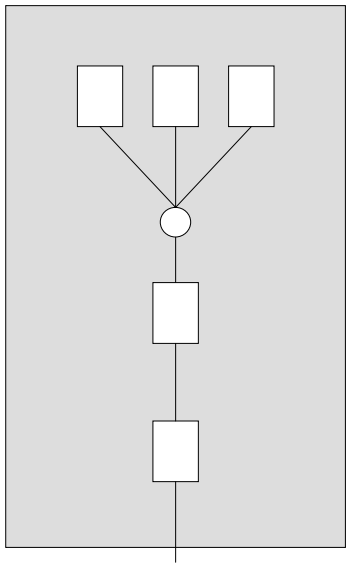
Multithreading

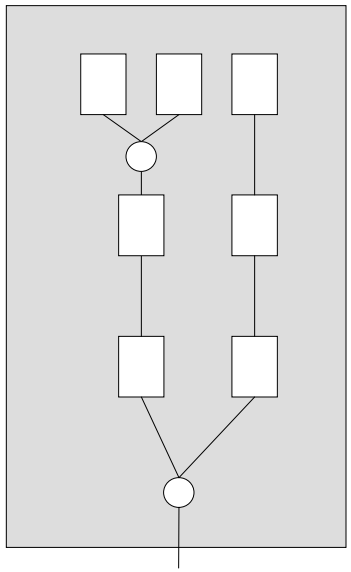


- Create one thread (process) per descriptor
- Threads can use regular blocking I/O

Multiplexing Discussion





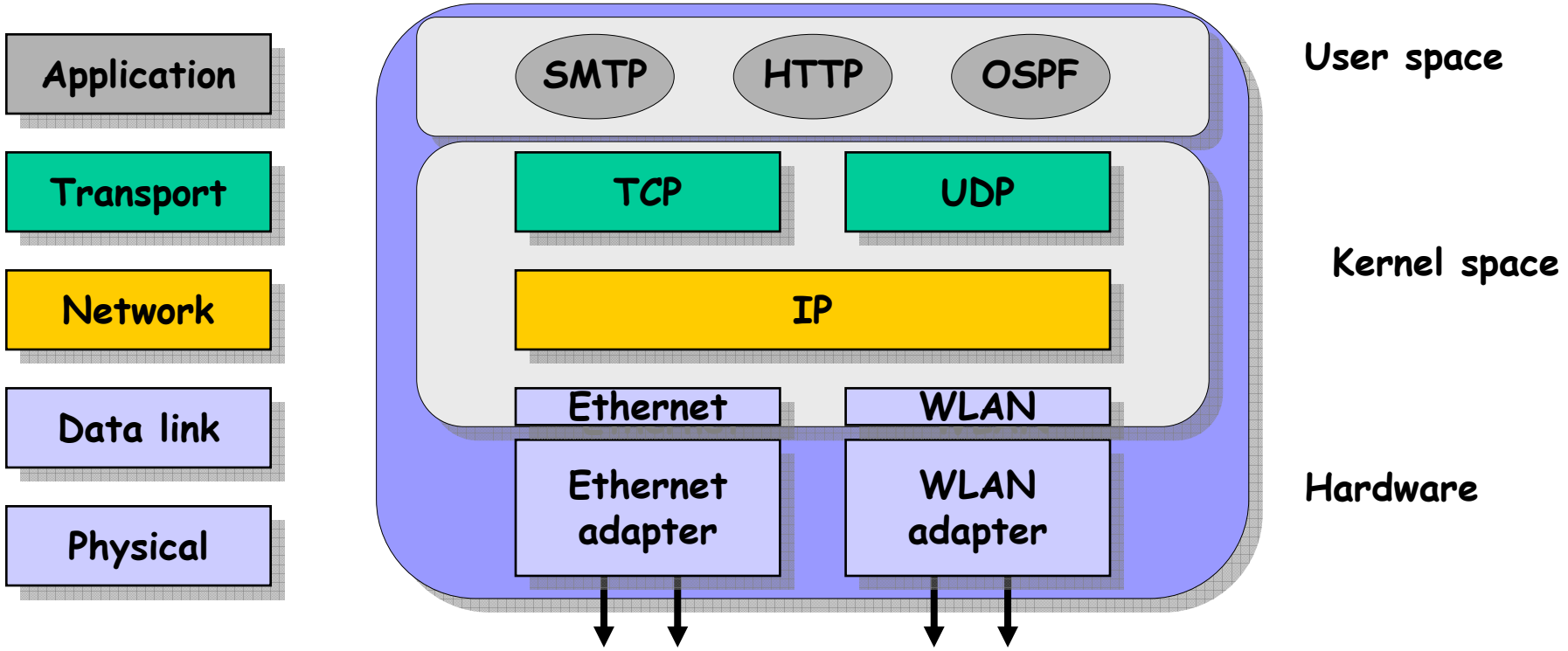


- There is a multiplexing point somewhere
  - One network interface, many applications
- Operating system or application
  - Or a combination of both

- "Best" choice may depend on your perspective
  - Software development
  - Application
  - System

Kernel Programming

- Transport layer and below reside in the kernel



Kernel Development

- UNIX kernel is a single binary file
- Loaded at boot time
- Development cycle is tedious
  - Change, compile, reboot
- Improved by loadable kernel modules
- Programming in an restricted environment
  - Learn, use and follow programming restrictions
    - Data structures, coding styles and conventions

Kernel Debugging

- Limited debugning facilities
  - printf (or printk)…
- Runtime errors produce crash and reboot
  - "panic: […]"
- Post-mortem analysis

Kernel Buffer Management

- Packet queues in kernel
  - Between network interface and kernel protocols
  - Between kernel protocols and application
- Shared pool of memory
  - Share memory in an efficient way between protocols, connections, etc
- Trade-off between memory efficiency and management overhead
  - Fixed-size buffers are easy to access, but waste memory
    - Need to be large enough for maximum size packets
  - Exact match requires more advanced memory management schemes
    - More processing and more control information
- Data structures to keep track of packet associated information together with buffer pointer
  - BSD "struct mbuf"
  - Linux "struct sk_buff"

Summary

- I/O models
  - Blocking
  - Non-blocking
  - Asynchronous
- Multiplexing models
  - Polling
  - Blocking
  - Multithreading
- Kernel programming and debugging