

Computer Networks

Adrian Sergiu DARABANT

Lecture 2

TCP Programming API

#include <sys/socket.h>

Client + Server :

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

Client:

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

Server:

int bind(int sfd, const **struct sockaddr** *servaddr, socklen_t addrlen);

int listen(int sockfd, int backlog);

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

Client + Server:

#include <unistd.h>

int close(int sockfd);

Socket Addresses - Generic

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

sa_data is a: (like when using **unions**)

```
struct sockaddr_in {  
    uint8_t sin_len; /* length of structure (16)*/  
    sa_family_t sin_family; /* AF_INET*/  
    in_port_t sin_port; /* 16 bit TCP or UDP port number */  
    struct in_addr sin_addr; /* 32 bit IPv4 address*/  
    char sin_zero[8]; /* not used but always set to zero - remaining to 14*/  
};
```

```
struct in_addr{  
    in_addr_t s_addr; /*32 bit IPv4 network byte ordered address*/  
};
```

TCP Programming API

Once a connection is setup:

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

```
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);  
int write(int sockfd, ..... ) – works as well
```

```
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);  
int read(int sockfd, ..... ) – works as well
```

TCP Programming



Client Server Iterative Example

TCP

◆ Connection-Oriented

- Recv - when RECV return N bytes have been received
- Send — when SEND returns N bytes have successfully delivered to OS and maybe to destination. Error will be signaled on next operation

◆ Guaranteed **data** delivery

◆ Guaranteed **data** ordering delivery

◆ Type = SOCK_STREAM — when creating socket

Client-Server TCP/IP Apps

- ◆ Server listens for client's requests, executes them and answers.
- ◆ Server types (from the Comm Point of View)
 - Iterative Servers (blocking)
 - Concurrent servers (blocking)(fork, threads)
 - Concurrent multiplexed servers. (**select**)

Concurrent TCP Servers

- Servers able to handle multiple clients in the same time;
- Implementation:
 - **Fork/CreateProcess** – independent client
 - **Threads** – interdependent states between clients
 - **Select** system call

Working with the DNS

```
struct hostent *gethostbyname(const char *name);  
struct hostent *gethostbyaddr(const void *addr, int len, int type);
```

hostent structure is defined in <netdb.h> as follows:

```
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 */  
}  
#define h_addr h_addr_list[0]
```

Another Approach: **getaddrinfo(...)**, **getnameinfo(...)**

Concurrent TCP Servers

Example – fork

Little Endian/Big endian

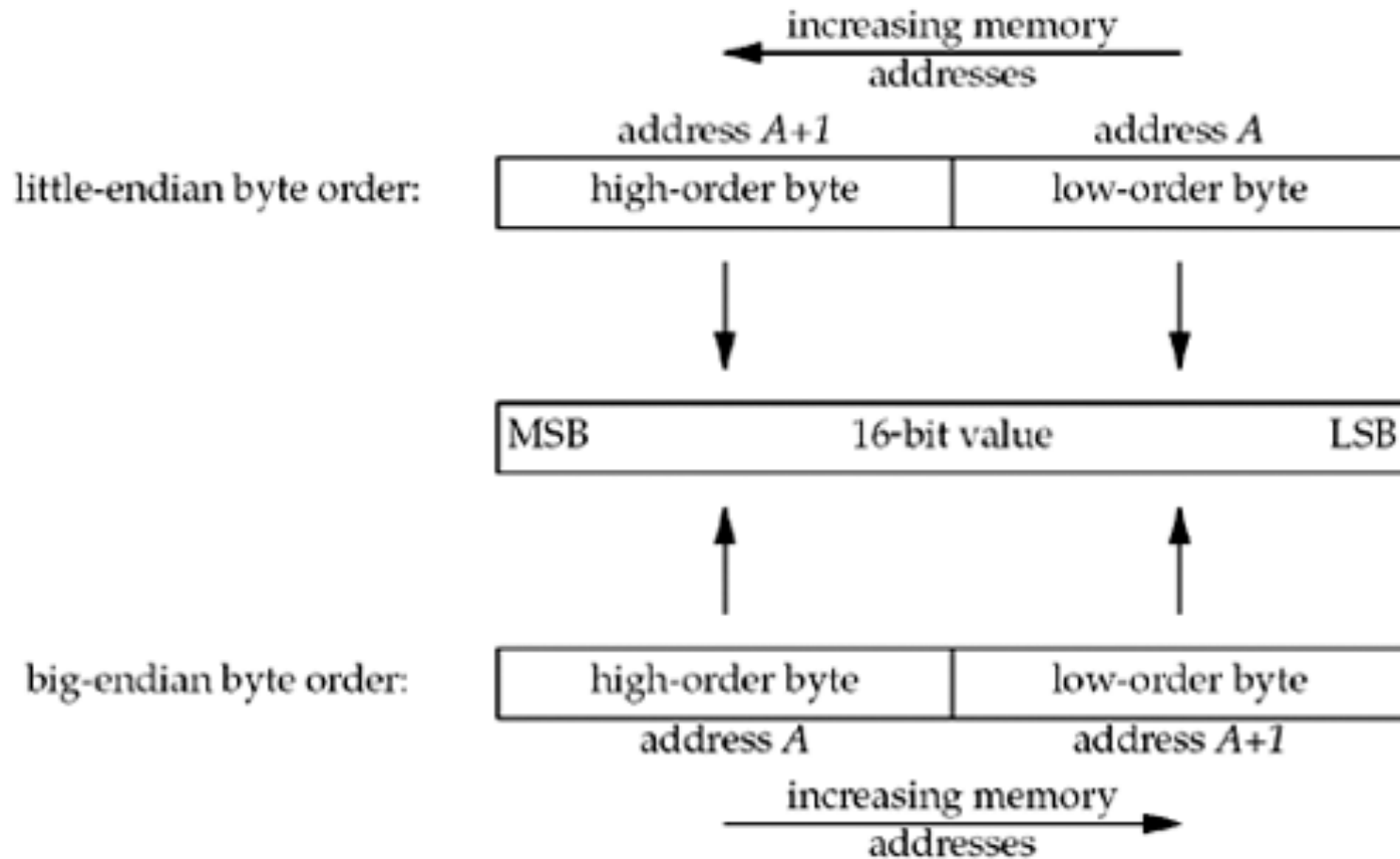
◆ In memory data representation

- Big endian – most significant byte first
- Little endian – least significant byte first

46F4 – Little end. => F4 46

- Big end. => 46 F4

Little Endian/Big endian



[illegible]

Double - 8 bytes

Swapping 8 or more byte entities ?!?

Example Concurrent Threaded Server

The server generates a random number. All connecting clients send a number to the server. The server answers with:

- 'S' – give a smaller number
- 'H' – give a larger number
- 'G' – you guessed my number
- 'L' – you lost. Somebody else guessed !

Write a client server implementation where each client is informed at the end of its status: winner/looser !

UDP

- ◆ Connection-Less – datagram oriented
 - Recvfrom - N bytes received
 - Sendto - N bytes given to the OS/network !!!
- ◆ No guarantee for datagram delivery
- ◆ No guarantee for datagram ordering
- ◆ Type = SOCK_DGRAM – when creating socket

TCP vs UDP

TCP

Write => stream of bytes.

Read=>reads from the stream

Bytes Not read from stream stay available for next read

Flow: neither party can overflow the other! Traffic is controlled by the OS

UDP

Write => packets of bytes.

Read=>reads bytes from one packet !

Not read bytes from a packet are lost !

Flow: one party can overflow the other =>lost packets ! No control !

UDP API

- `socket(..., SOCK_DGRAM,...)`
- `bind` – similar to TCP - has the same meaning
- Data exchange – `sendto`, `recvfrom`
- `close` – same meaning as TCP
- **Remark:**
 - No listen and accept are required/permitted !

UDP API

ssize_t **sendto**(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);

Return:

- no of bytes transmitted to the OS/network if positive
- -1 in case of locally detected error (errno set appropriately)
- No error signaling for undelivered data !
- Data size must fit into a transmission unit(datagram) < 64Kb or Error !
- One sendto operation must be consumed by exactly one recvfrom operation on the receiver!

UDP API

ssize_t **recvfrom**(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);

Return:

- no of bytes received from the OS/network and stored into **buf**
- -1 in case of error (errno set appropriately)
- Call blocks until a datagram arrives
- If not all data is read in ONE call – the remaining is discarded
- Cannot receive more than the maximal transfer unit (datagram) size – max. 64 Kb

Socket Options

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

getsockopt(.....)

man 7 socket/ip/udp/tcp

Optname

SO_REUSEADDR – reuse local addresses – use it to get rid of the address already in use after you terminate your TCP/IP app abruptly !

SO_BROADCAST – enables broadcast = one sender – Universal(all) receiver(s)!

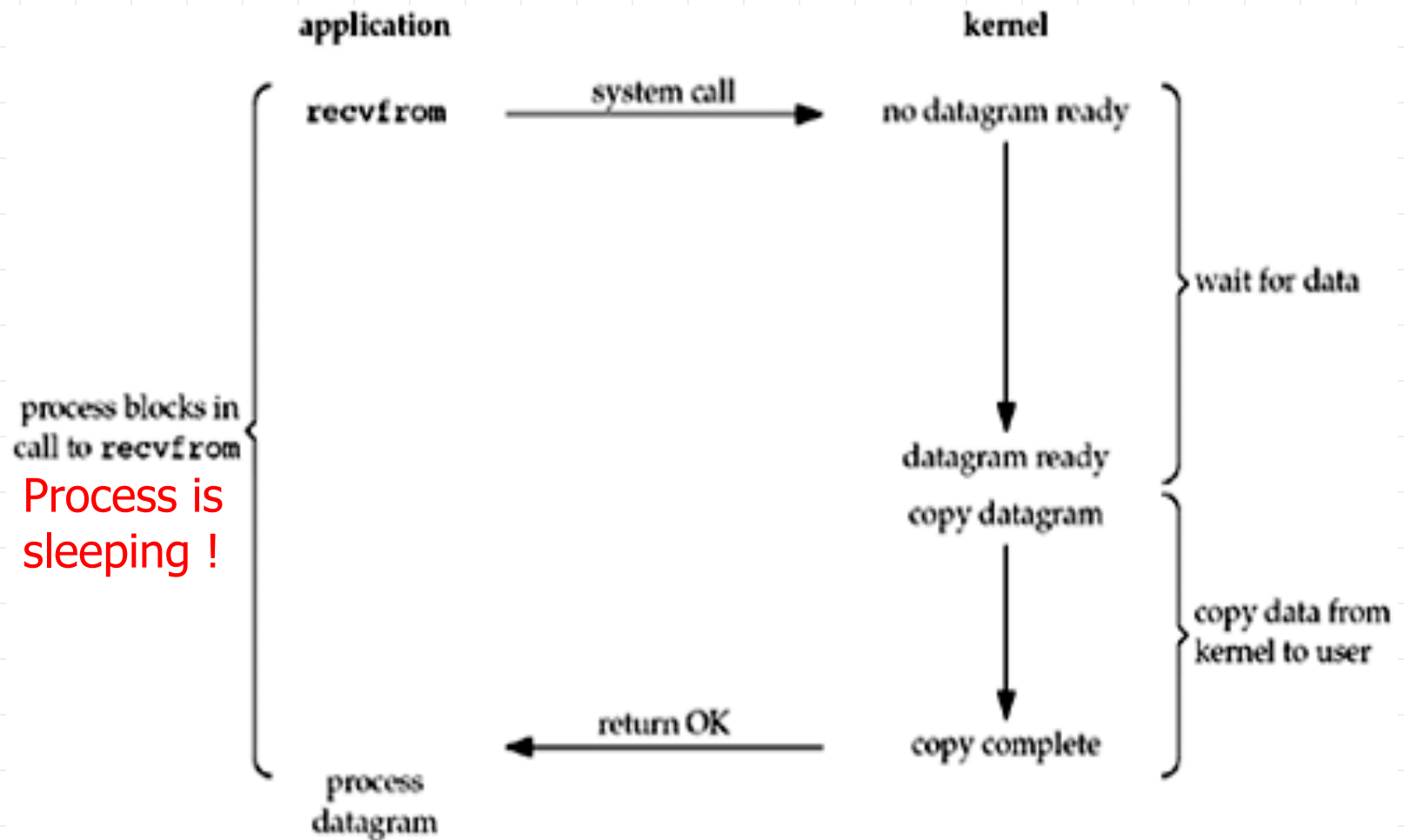
Advanced TCP/IP – I/O Modes

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

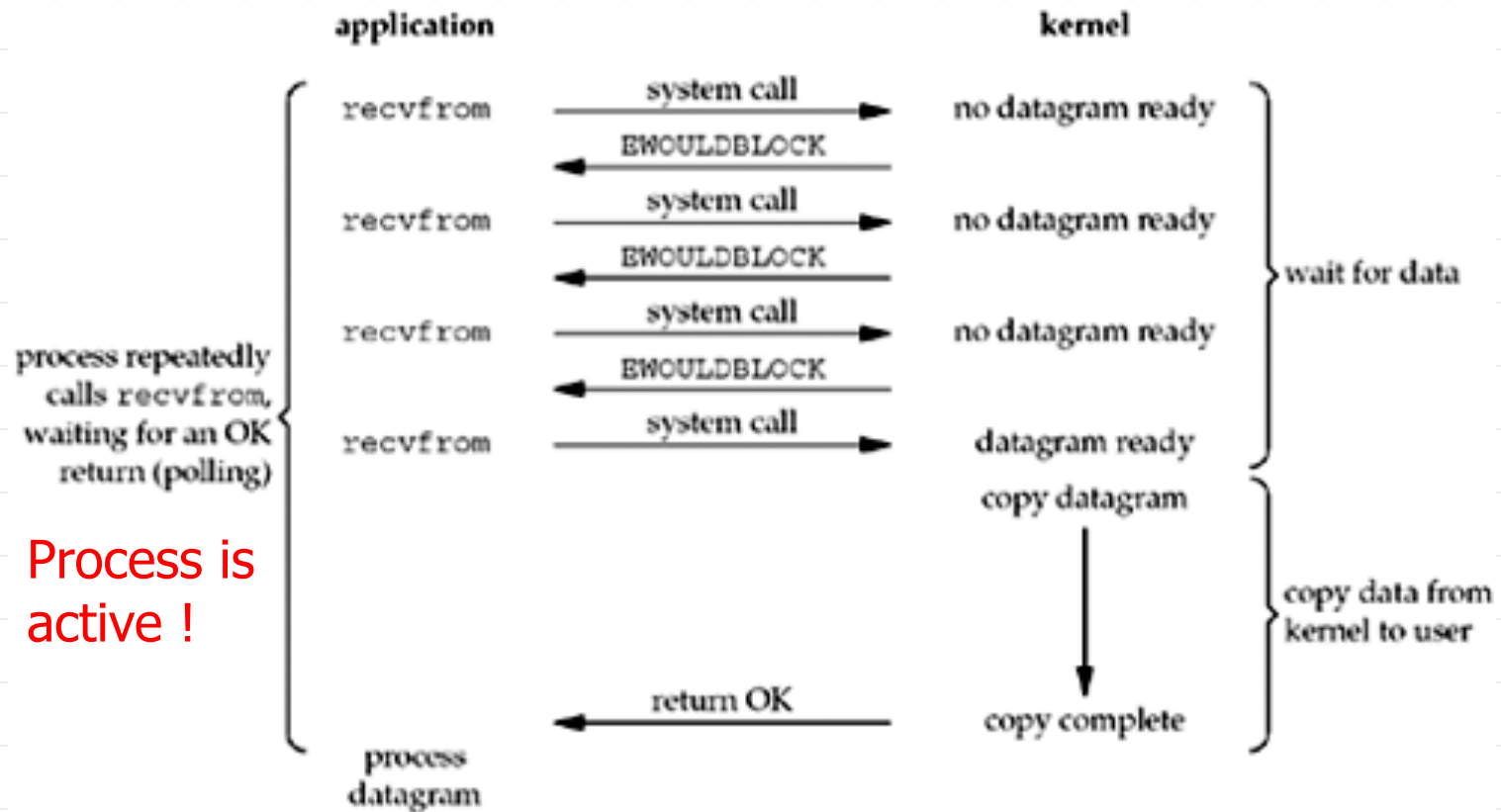
Network operation steps - READ

1. Waiting for the data to be ready – i.e. data arrives from the network
2. Copying the data from the kernel to the process – i.e. copying the data from the kernel buffer into the application space.

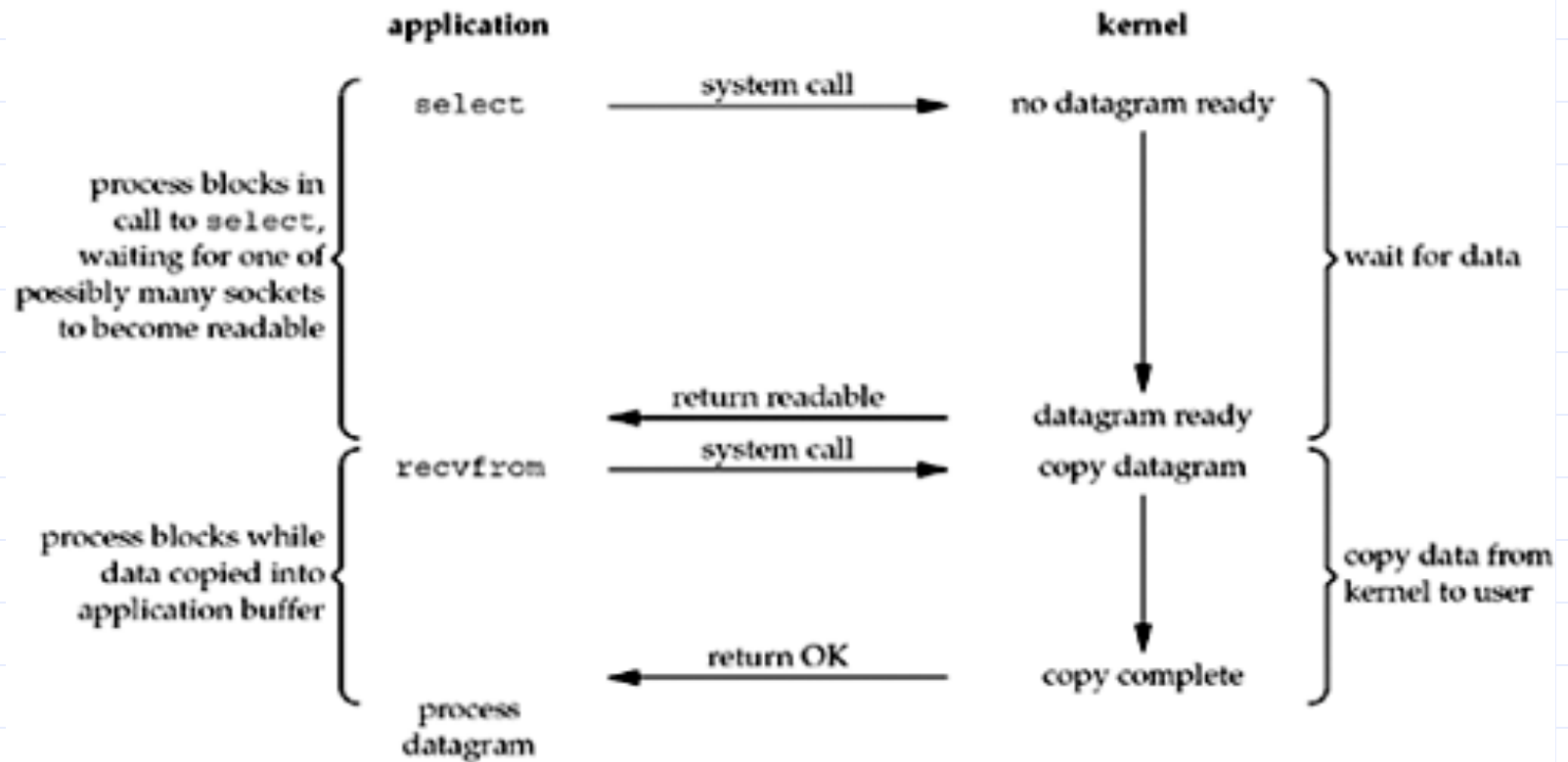
Blocking I/O Model



Non-Blocking I/O Model

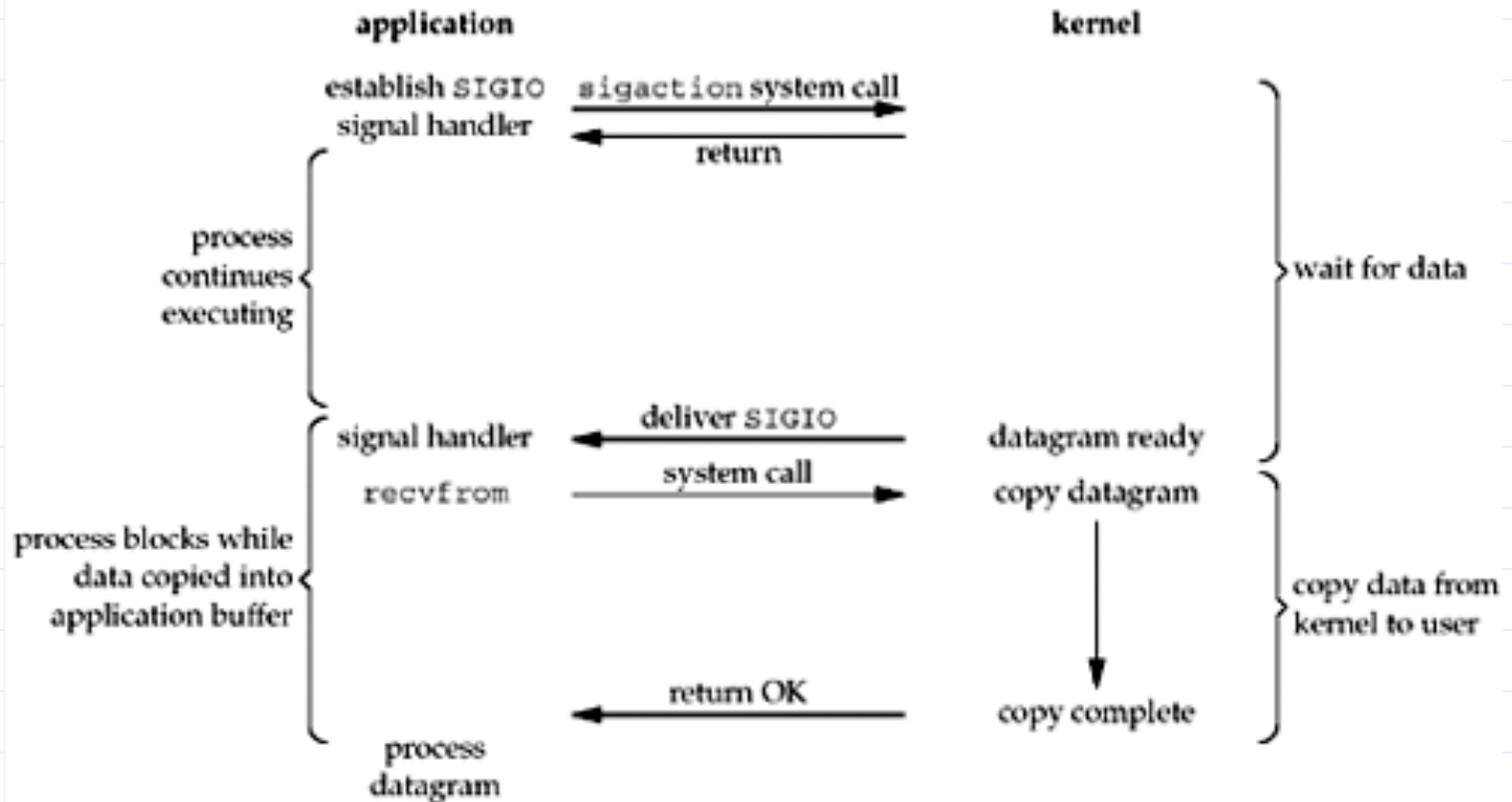


Multiplexed I/O Model

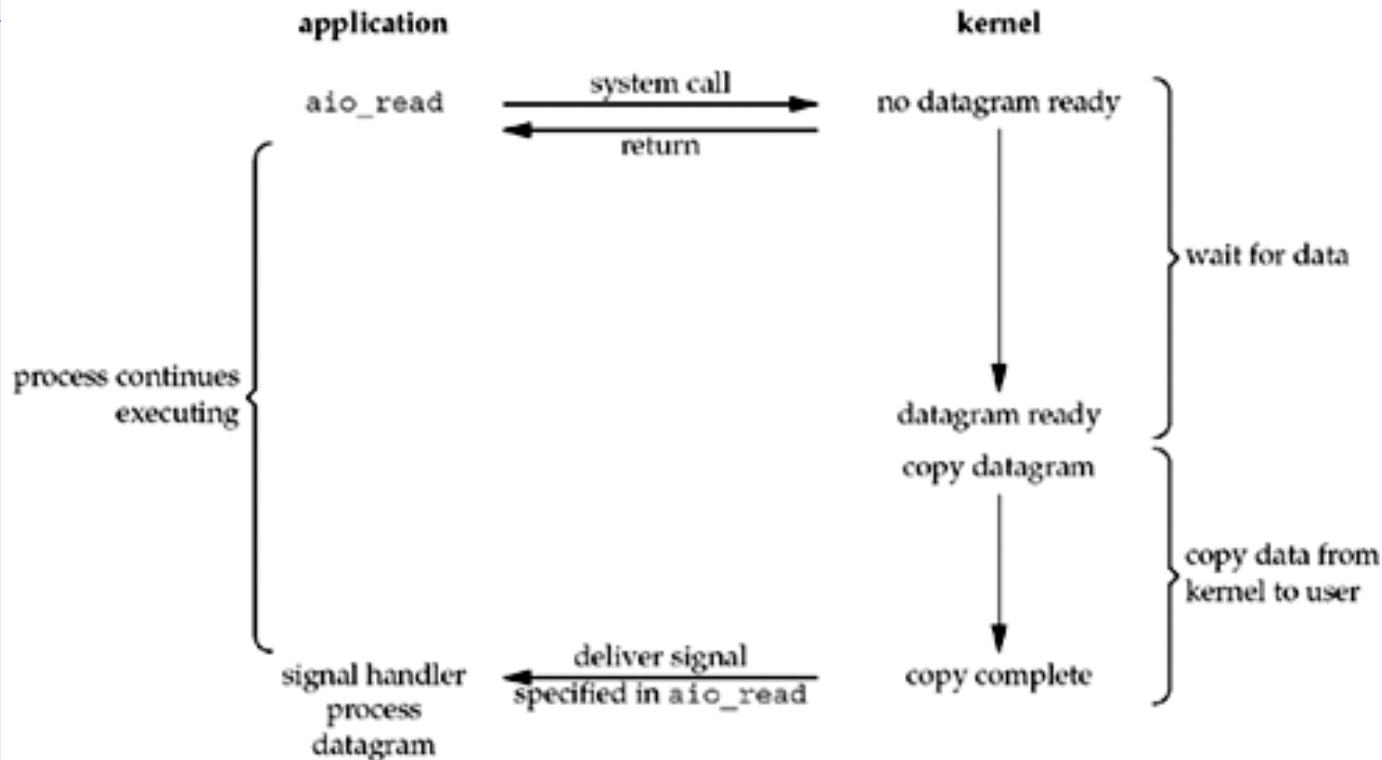


Involves using select and poll

Signal I/O Model



Asynchronous I/O Model



- **aio_read – asynchronous POSIX read**
- Async - Kernel tells us when the operation is complete
- Signal – Kernel tells us when the operation can start !!

Blocking I/O Operations Sequence

Client

Write(...)
Write(...)
Read(...)

?

Server

Read(...)
Read(...)
Write(...)

Write(...)
Write(...)
Read(...)

?

Write(...)
Read(...)
Write(...)

Read(...)
Write(...)
Read(...)

?

Read(...)
Read(...)
Write(...)

The **select** system call

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

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

```
int select(int maxfd+1, fd_set *readset, fd_set *writeset, fd_set *exceptset,  
           const struct timeval *timeout);
```

Returns:

- positive count of ready descriptors
- 0 if **timeout**
- -1 on error

◆ void FD_ZERO(fd_set *fdset);

- clear all bits in fdset

◆ void FD_SET(int fd, fd_set *fdset);

- turn on the bit for fd in fdset

◆ void FD_CLR(int fd, fd_set *fdset);

- turn off the bit for fd in fdset

◆ int FD_ISSET(int fd, fd_set *fdset);

- IS the fd ready ?

◆ **BE WARNED** – select modifies readset, writeset and exceptset

Conditions for a socket to be ready

<u>Condition</u>	<u>Readable</u>	<u>Writable</u>	<u>Exception</u>
Data to read	Y		
Read half connection closed	Y		
New connection (for listen)	Y		
Space available for writing		Y	
Write half connection closed		Y	
Pending error	Y	Y	
TCP- out-of-bound DATA			Y

MultiClient Chat Server -1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
fd_set master; // master file descriptor list
fd_set read_fds; // temp file descriptor list for
    select()
struct sockaddr_in myaddr; // server address
struct sockaddr_in remoteaddr;
```

```
int fdmax; // maximum file desc. number
int listener; // listening socket descriptor
int newfd; // newly accept()ed socket
char buf[256], tmpbuf[256];
int nbytes, ret;
int yes=1; // setsockopt() SO_REUSEADDR
int addrlen;
int i, j, crt, int_port, client_count=0;
```

MultiClient Chat Server -2

```
struct sockaddr_in getSocketName(int s, bool local_or_remote) {  
    struct sockaddr_in addr;  
    int addrlen = sizeof(addr);  
    int ret;  
  
    memset(&addr, 0, sizeof(addr));  
    ret = (local_or_remote==true?getsockname(s,(struct sockaddr *)&addr,  
                                              (socklen_t*)&addrlen):  
getpeername(s,(struct sockaddr *)&addr, (socklen_t*)&addrlen) );  
    if (ret < 0)  
        perror("getsock(peer)name");  
    return addr;  
}
```


MultiClient Chat Server -3

```
char * getIPAddress(int s, bool local_or_remote) {  
    struct sockaddr_in addr;  
    addr = getSocketName(s, local_or_remote);  
    return inet_ntoa(addr.sin_addr);  
}
```

```
int getPort(int s, bool local_or_remote) {  
    struct sockaddr_in addr;  
    addr = getSocketName(s, local_or_remote);  
    return addr.sin_port;  
}
```


MultiClient Chat Server -5

```
int main(int argc, char **argv) {
    if (argc < 2 ) {
        printf("Usage:\n%s <portno>\n",argv[0]);
        exit(1);
    }
    int_port = atoi(argv[1]);
    FD_ZERO(&master); // clear the master and temp sets
    FD_ZERO(&read_fds);
    // get the listener
    if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
    // get rid of the "address already in use" error message
    if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int) ) == -1) {
        perror("setsockopt:");
        exit(1);
    }
}
```


MultiClient Chat Server -7

```
// main loop
```

```
for(;;) {
```

```
    read_fds = master;
```

```
// copy it – select
```

```
    if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
```

```
        perror("select");
```

```
        exit(1);
```

```
    }
```

```
// run through the existing connections looking for data to read
```

```
for(i = 0; i <= fdmax; i++) {
```

```
    if (FD_ISSET(i, &read_fds)) {           // we got one!!
```

```
        crt = i;
```

```
        if (i == listener) {
```

```
            // handle new connections
```

```
            addrlen = sizeof(remoteaddr);
```

```
            if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr, (socklen_t*)& addrlen)) == -1) {
```

```
                perror("accept");
```

```
            }
```

MultiClient Chat Server -8

```
else {
    FD_SET(newfd, &master); // add to master set
    if (newfd > fdmax) { // keep track of the maximum
        fdmax = newfd;
    }
    printf("selectserver: new connection from %s on socket %d\n",
           getIPAddress(newfd, false), newfd);

    client_count++;
    sprintf(buf, "Hi-you are client :[%d] (%s:%d) connected to server %s\nThere
are %d clients connected\n", newfd, getIPAddress(newfd, false), getPort(newfd, false),
getIPAddress(listener, true), client_count);
    send(newfd, buf, strlen(buf)+1, 0);
}
}
```

MultiClient Chat Server -9

```
else {
    // handle data from a client
    if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
        // got error or connection closed by client
        if (nbytes == 0) {
            // connection closed
            printf("<selectserver>: client %d forcibly hung up\n", i);
        }
        else perror("recv");
        client_count--;
        close(i); // bye!
        FD_CLR(i, &master); // remove from master set
    }
    else {
        // we got some data from a client - check for connection close request
        buf[nbytes]=0;
        if ( (strncasecmp("QUIT\n",buf,4) == 0)) {
            sprintf(buf,"Request granted [%d] - %s. Disconnecting...\n",i,getIPAddress(i,false));
```

MultiClient Chat Server -10

```
    send(i,buf, strlen(buf)+1,0);
    nbytes = sprintf(tmpbuf,"<%s - [%d]> disconnected\n",getIPAddress(i,false), i);
    sendToALL(tmpbuf,nbytes);
    client_count--;
    close(i);
    FD_CLR(i,&master);
}
else {
    nbytes = sprintf(tmpbuf, "<%s - [%d]> %s",getIPAddress(crt, false),crt, buf);
    sendToALL(tmpbuf, nbytes);
}
}
}
}
}
}
}
return 0;
}
```


MultiClient Chat Client -1

```
....//include stuff
```

```
.....
```

```
fd_set read_fds, master; // temp file descriptor list for select()
```

```
int sock;      //socket
```

```
struct sockaddr_in servaddr;
```

```
char buf[256]; // buffer for client data
```

```
int nbytes, ret, int_port;
```

```
int main(int argc, char **argv)
```

```
{
```

```
    if (argc < 3 ) {
```

```
        printf("Usage:\n%s <hostname or IP address> <portno>\n", argv[0]);
```

```
        exit(1);
```

```
    }
```

MultiClient Chat Client -2

```
int_port = atoi(argv[2]);
int ipaddr = inet_addr(argv[1]);

// check if address is a hostname
if (ipaddr == -1 ) {
    struct in_addr inaddr;
    struct hostent * host = gethostbyname( argv[1] );
    if (host == NULL ) {
        printf("Error getting the host address\n");
        exit(1);
    }
    memcpy(&inaddr.s_addr, host->h_addr_list[0], sizeof(inaddr));
    printf("Connecting to %s ...\n", inet_ntoa( inaddr ) );
    memcpy(&ipaddr, host->h_addr_list[0], sizeof(unsigned long int)) ;
}

// create the socket
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
}
```

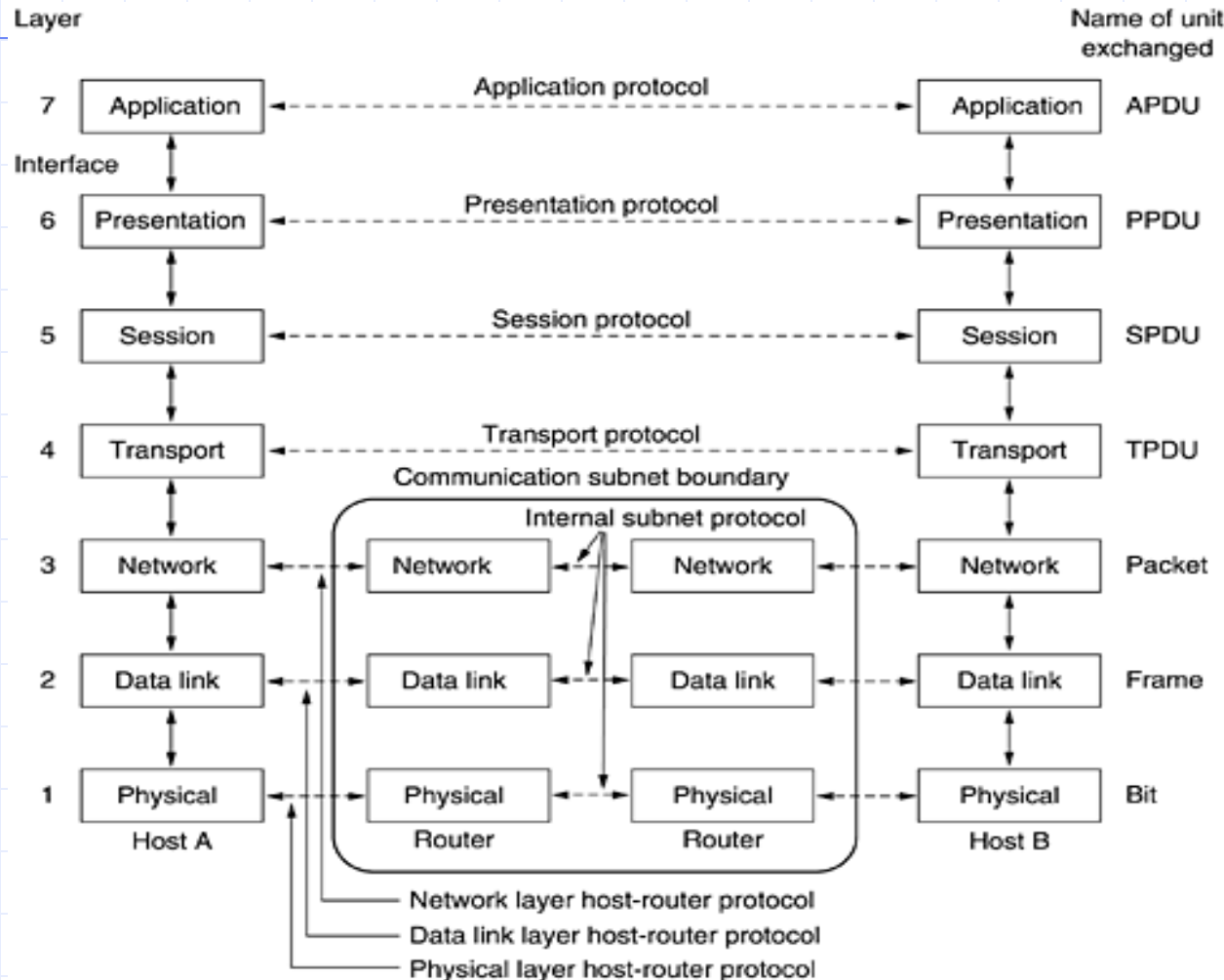
MultiClient Chat Client -3

```
memset(&servaddr,0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = ipaddr;
servaddr.sin_port = htons( int_port );
// connect to server
if (connect(sock, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0 ) {
    perror("connect");
    exit(1);
}
// add the socket to the master set
FD_ZERO(&read_fds); // clear the set
FD_ZERO(&master);
FD_SET(0, &master); FD_SET(sock, &master);
while(1) {
    read_fds = master;
    if (select(sock+1, &read_fds, NULL, NULL, NULL) == -1) {
        perror("select");
        exit(1);
    }
}
```

MultiClient Chat Client -4

```
if ( FD_ISSET(0, &read_fds) ) {                                // check if read from keyboard
    nbytes = read(0, buf, sizeof(buf)-1);
    ret = send(sock, buf, nbytes, 0);
    if (ret <= 0 ) {
        perror("send");
        exit(1);
    }
}
if ( FD_ISSET(sock, &read_fds) ) {                               // check if read from server
    nbytes = read(sock, buf, sizeof(buf)-1);
    if (nbytes <= 0) {
        printf("Server has closed connection... closing...\n");
        exit(2);
    }
    write(1, buf, nbytes);
}
}
return 0;
}
```

The OSI Reference Model



All People Seem To Need Data Processing

Principles of the OSI model

1. A layer should be created where a different abstraction is needed.
2. Each layer should perform a well-defined function.
3. The function of each layer should be chosen with an eye toward defining internationally standardized protocols.
4. The layer boundaries should be chosen to minimize the information flow across the interfaces.
5. The number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity and small enough that the architecture does not become unwieldy.

The Physical Layer

- ◆ Raw bits over a communication channel
- ◆ Data representation
 - 1—how many volts ?; 0 – how many volts ?
- ◆ 1 bit – How many nanoseconds ?
- ◆ Bidirectional simultaneous transmission?
- ◆ Electrical, mechanical, timing interfaces

Data Link layer

- ◆ Turn the raw transmission into an error free communication line
- ◆ Sets data in **frames**=thousands of bytes
- ◆ Traffic regulation (flow control)
- ◆ Access to the medium in broadcast shared communication lines

The Network Layer

- ◆ Controls the operation of a subnet
- ◆ How packets are routed from source to destination
- ◆ Quality of service – congestion control
- ◆ Fragmentation and inter-network problems

The Transport Layer

- ◆ Accept data from upper layers and splits it into packets (small units)
- ◆ Ensure that packets arrive correctly to the other end
- ◆ Type of service: error free PtoP, preserve order or not, guarantees delivery or not, broadcast
- ◆ True end-to-end layer

The Session Layer

◆ Allows for establishing sessions

◆ Session

- Dialog control
- Token management
- Synchronization

The Presentation Layer

- ◆ Syntax and semantics of data
- ◆ Abstract data definitions/ encoding for information exchange between heterogeneous systems
- ◆ Standard encoding “on the wire”
- ◆ Exchange unit – record type

The Application Layer

◆ Protocols needed by users:

- HTTP - www
- FTP – file exchange
- TELNET – remote command
- SSH – remote command
- SMTP – mail exchange