Computer Networks

Adrian Sergiu DARABANT

Lecture 2

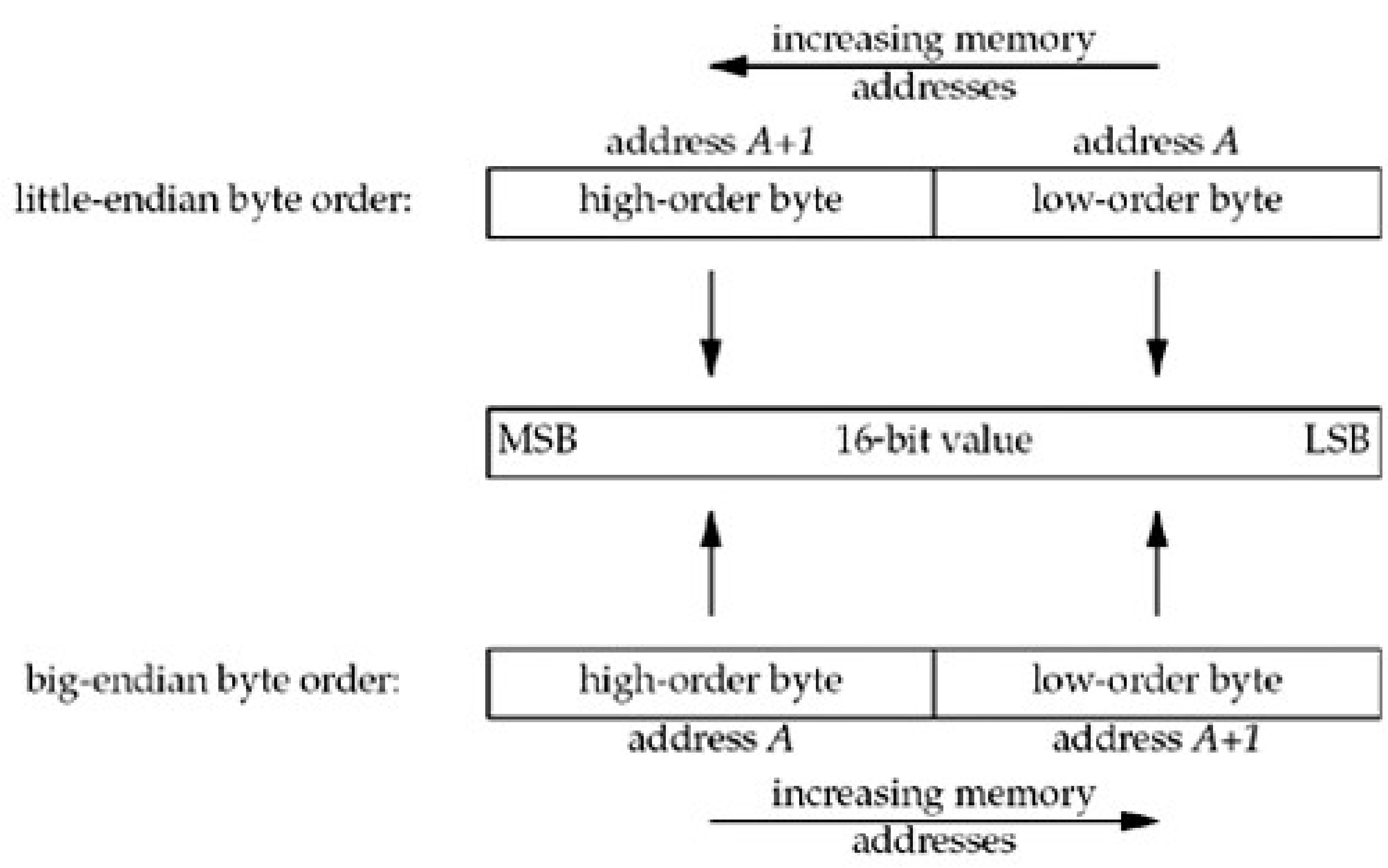
Little Endian/Big endian

- 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



Float and double

- Float – 4 bytes

31																															0
s	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

exp mantisa

- Float = (sign?-1:1) * 2^exp *1.<mantisa>
- Double - 8 bytes

- **<u>Same endianness</u>** as integers
- Swapping 8 or more byte entities ?!?

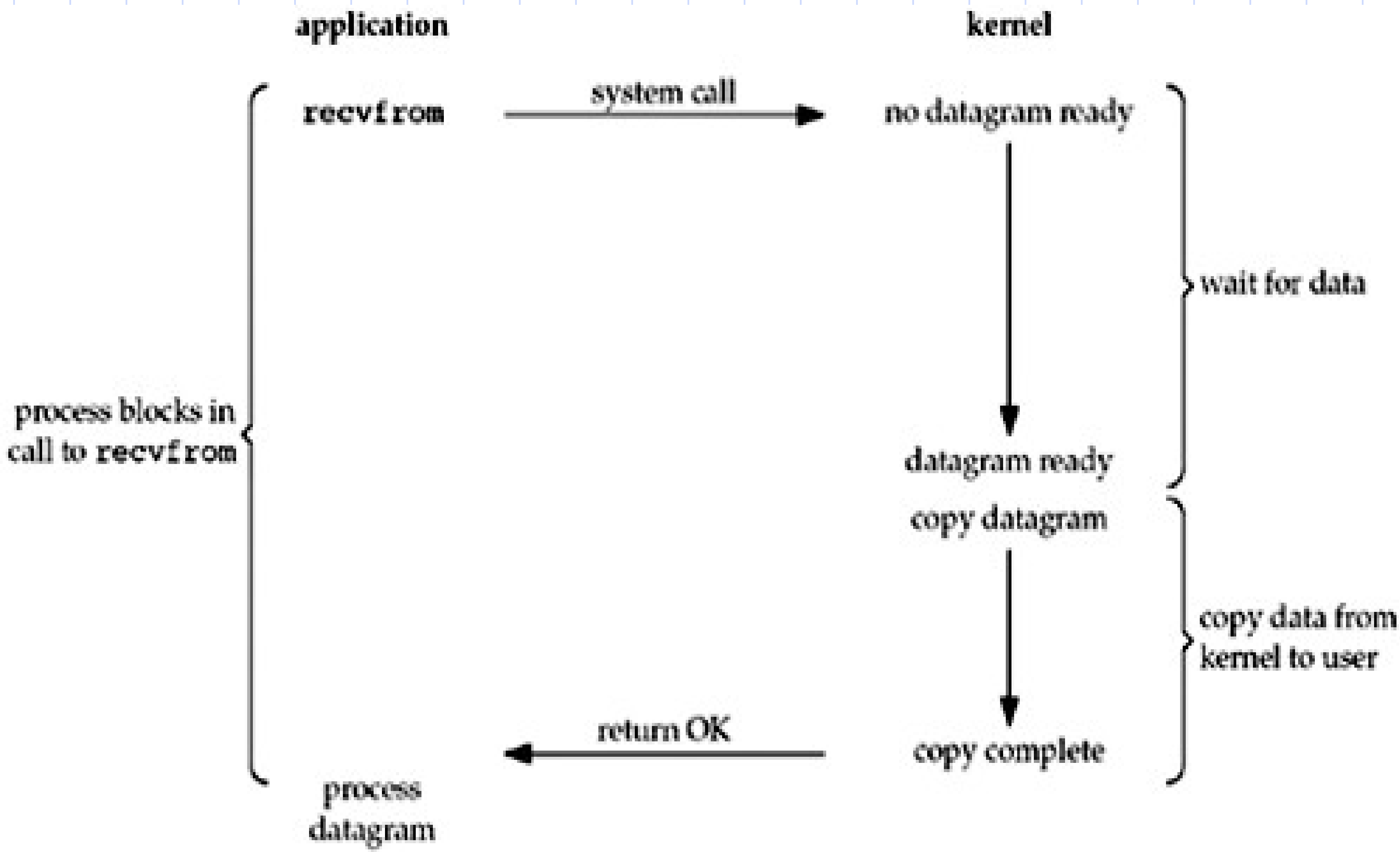
Advanced TCP – I/O Modes

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)

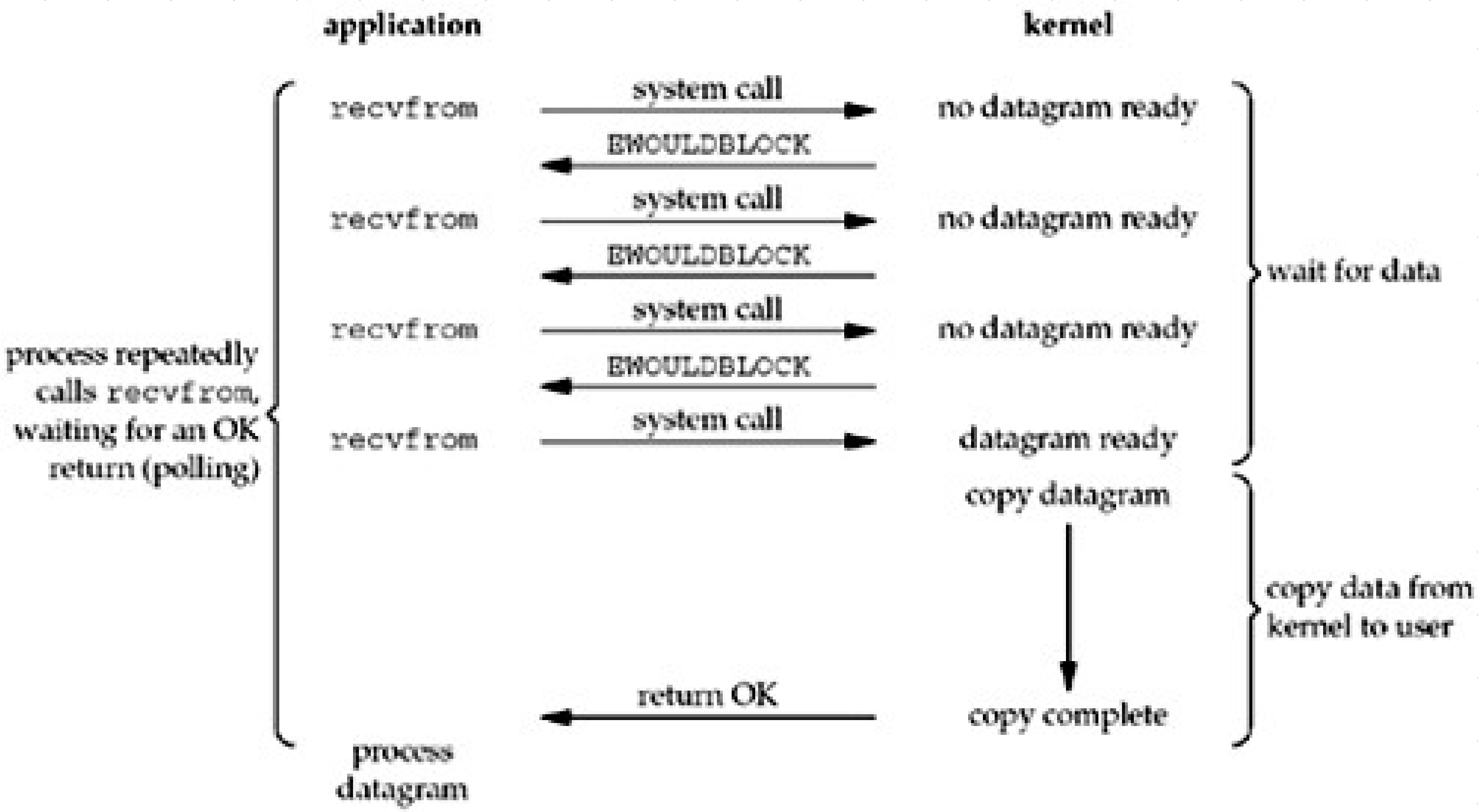
Network operation steps

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.

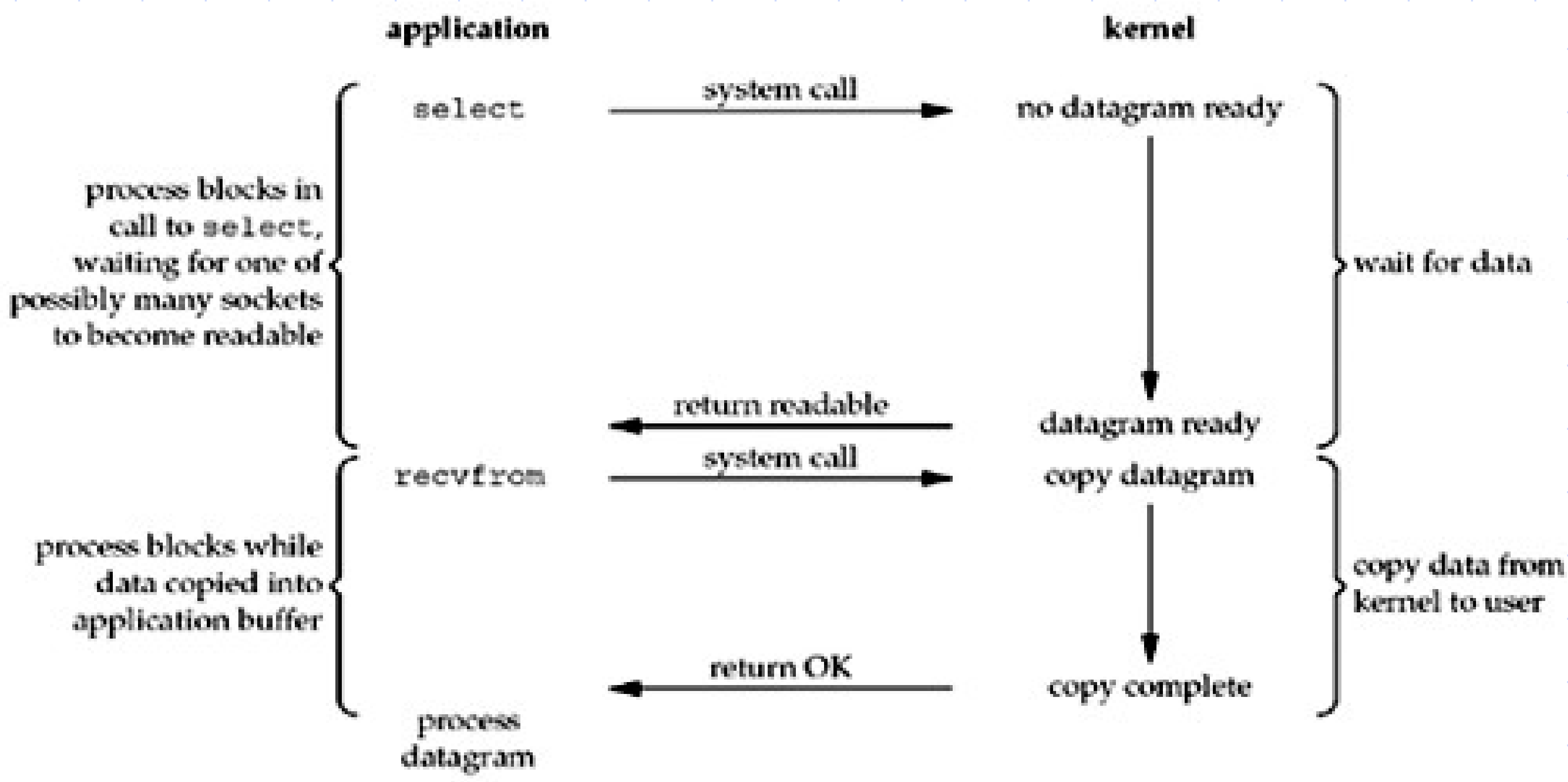
Blocking I/O Model



Non-Blocking I/O Model

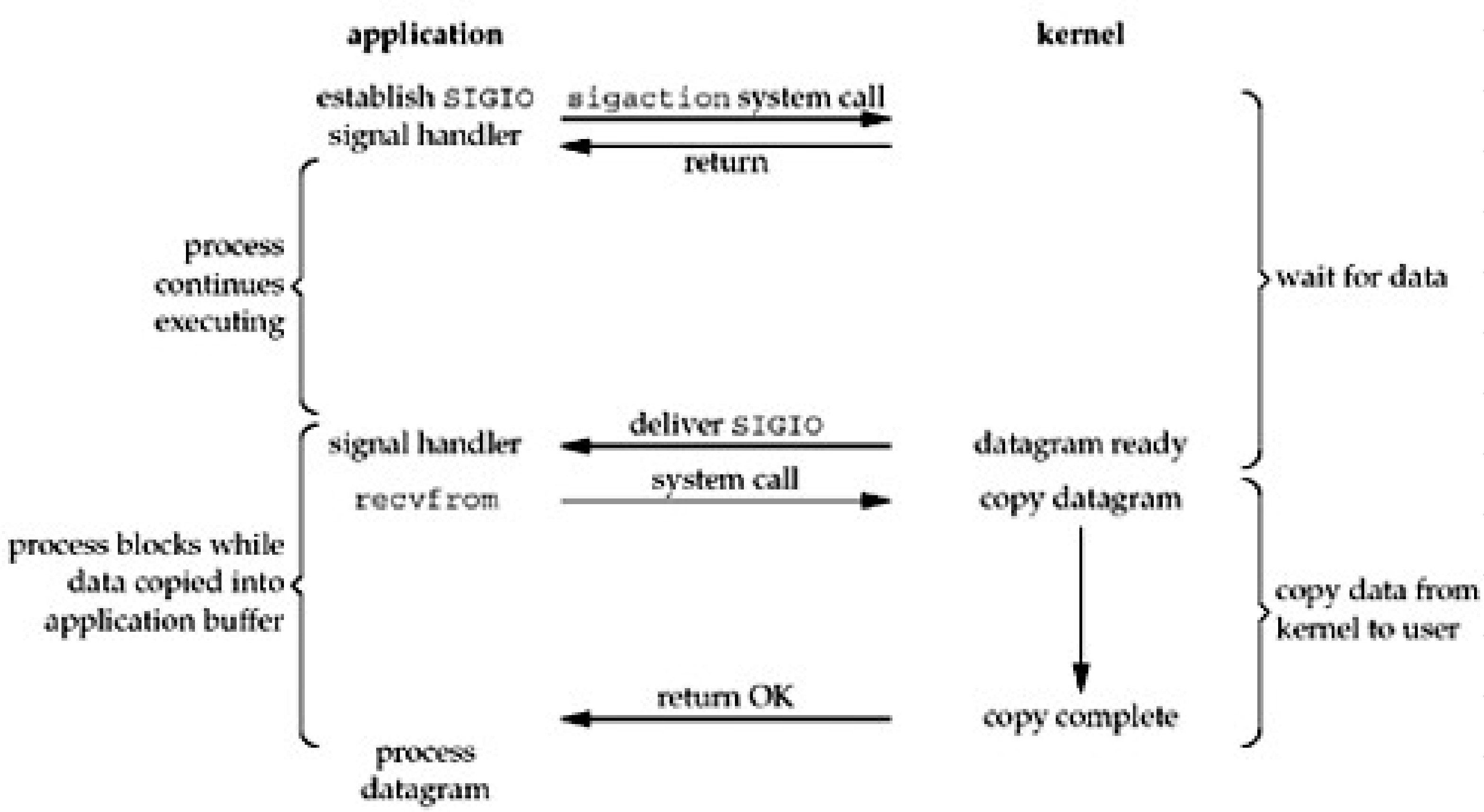


Multiplexed I/O Model

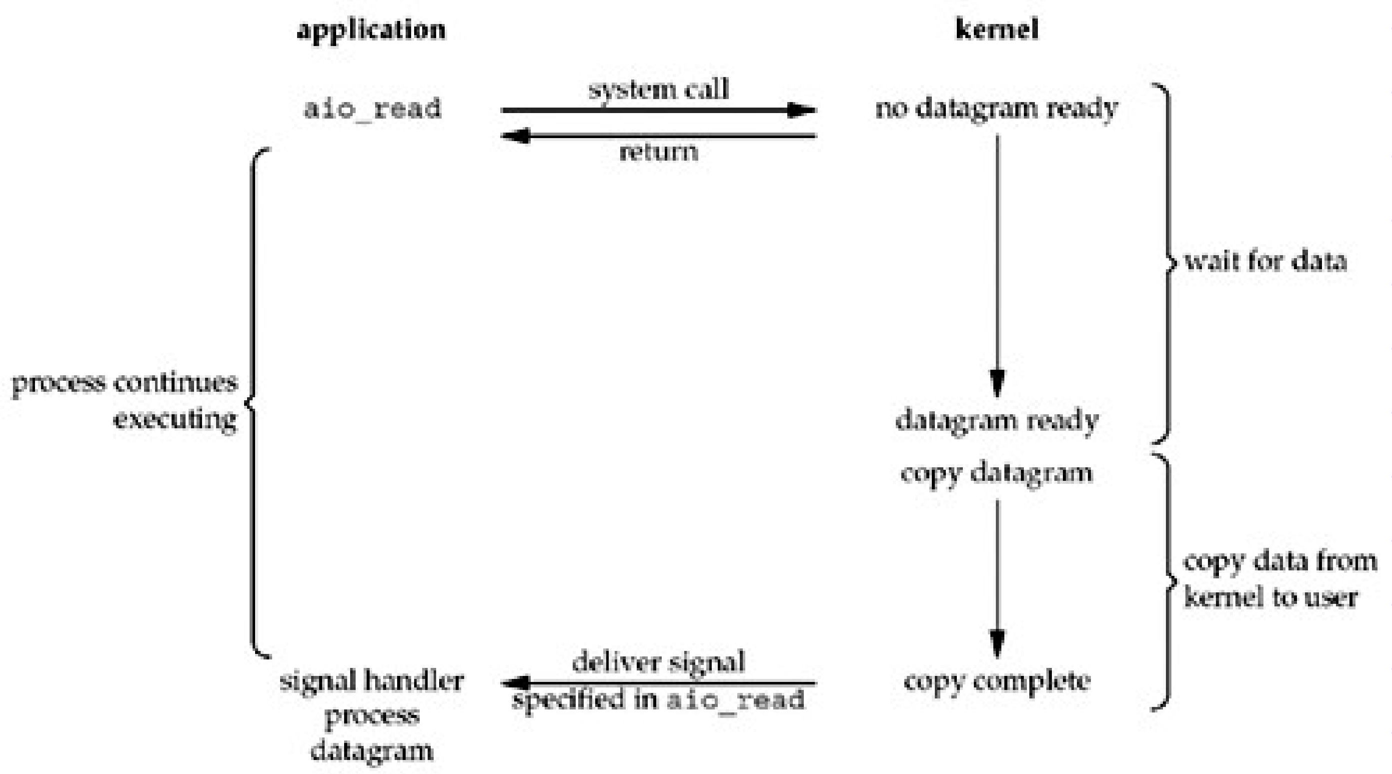


Involves using *select* and *poll*

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

Blocking I/O Operations Sequence

Client		Server
Write(…) Write(…) Read(…)	?	Read(…) Read(…) Write(…)
Write(…) Write(…) Read(…)	?	Write(…) Read(…) Write(…)
Read(…) Write(…) Read(…)	?	Read(…) Read(…) Write(…)

TCP

- Connection-Oriented
  - Recv
  - Send
- Guaranteed **data** delivery
- Guaranteed **data** ordering delivery
- Type = SOCK_STREAM – when creating socket

UDP

- Connection-Less – datagram oriented
  - Recvfrom
  - Sendto
- No guarantee for datagram delivery
- No guarantee for datagram ordering
- Type = SOCK_DGRAM – when creating socket

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

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

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 (fork, threads)
  - Concurrent multiplexed servers. (select)

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(….)**

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 -4

```
// send to everyone
void sendToALL(char * buf, int nbytes) {
  int j, ret;
  for(j = 0; j <= fdmax; j++) {
    if (FD_ISSET(j, &master))
      // except the listener and ourselves
      if (j != listener && j != crt)
     if ( send(j, buf, nbytes, 0) == -1)
       perror("send");
  }
  return;
}
```

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 -6

```
// bind
    memset(&myaddr, 0, sizeof(myaddr));
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = INADDR_ANY;
    myaddr.sin_port = htons(int_port);
    if (bind(listener, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
        perror("bind:");
        exit(1);
    }
    // listen
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(1);
    }
    // add the listener to the master set
    FD_SET(listener, &master);
    // keep track of the biggest file descriptor
    fdmax = listener;                                   // so far, it's this one
```

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;
}
```

Socket Options

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

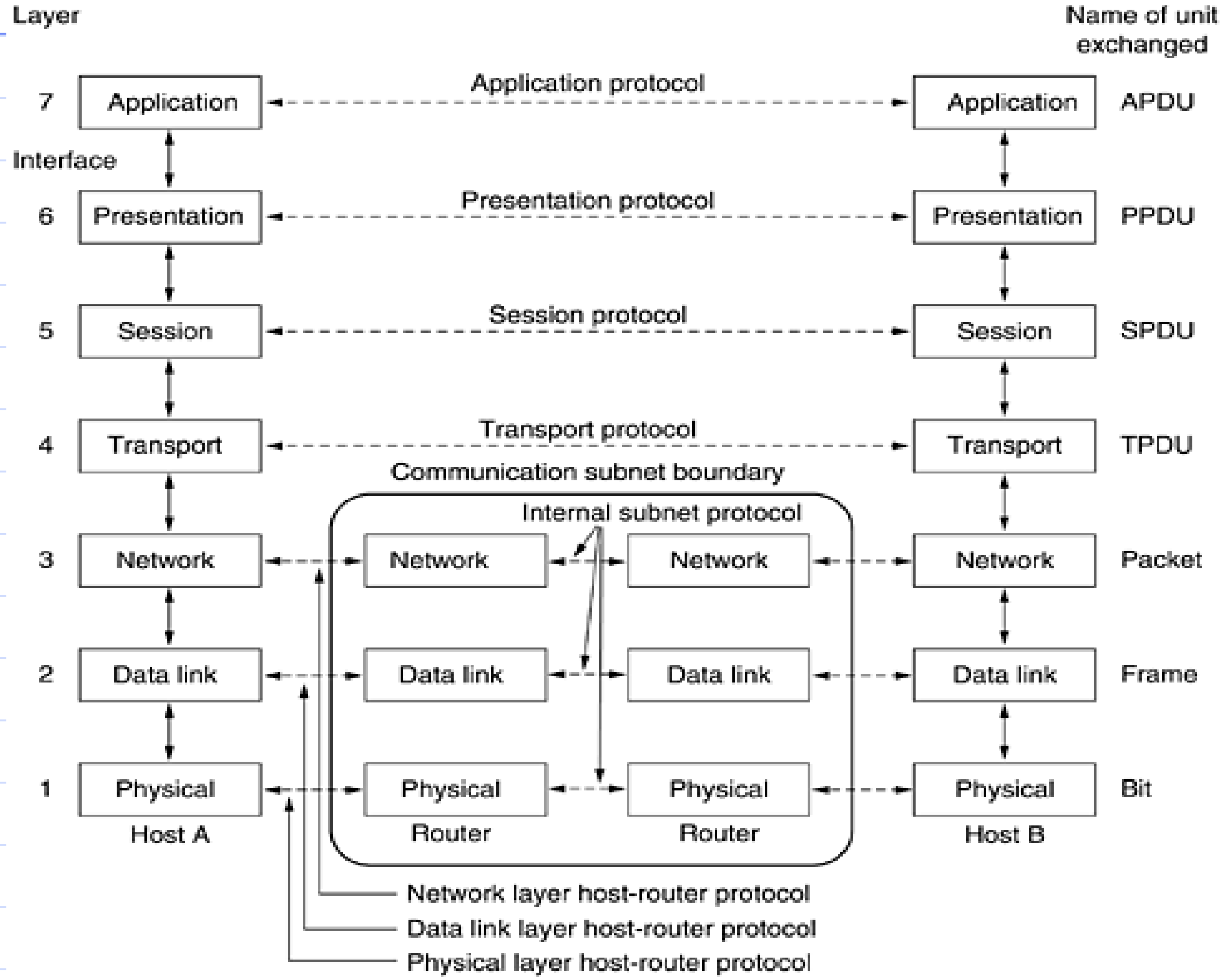
getsockopt(.....)

Optname

SO_REUSEADDR – reuse local addresses

SO_BROADCAST – enables broadcast

The OSI Reference Model



All **P**eople **S**eem **T**o **N**eed **D**ata **P**rocessing

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