Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Outline

Outline

- Advanced I/O Functions
  - How much data is queued?
  - Sockets and standard I/O
- Unix Domain Sockets
  - Socket pair
  - Stream sockets
  - Datagram sockets
  - Passing descriptors
  - Passing Credentials
- Unix I/O Models
  - Blocking
  - Non-blocking
  - I/O multiplexing
  - Signal driven
  - Asynchronous
- I/O Multiplexing
  - select()
  - Client using select()
  - Concurrent server using select()

innovate achieve lead

BITS Pilani
Pilani Campus

Advanced I/O

T1: 14.7 - 14.8

How much data is Queued?

- Use *recv*() with MSG_PEEK flag.

```
int numbytes = recv(fd, buf, bufsize, MSG_PEEK);
```

  - For TCP, this will give the number of bytes available in socket recv buffer.
    - This value could change in between two reads.
  - For a connected UDP socket, this return the number of bytes in the next available datagram.
    - Between two reads this value remains same.
- Use *ioctl*() call with FIONREAD command.

```
ioctl(fd, FIONREAD, &numbytes)
```

- In UDP case, the size of datagram can be zero. This makes it difficult to distinguish between nodata or data.
  - Safer to use *select*() first and then call I/O.

Sockets and Standard I/O

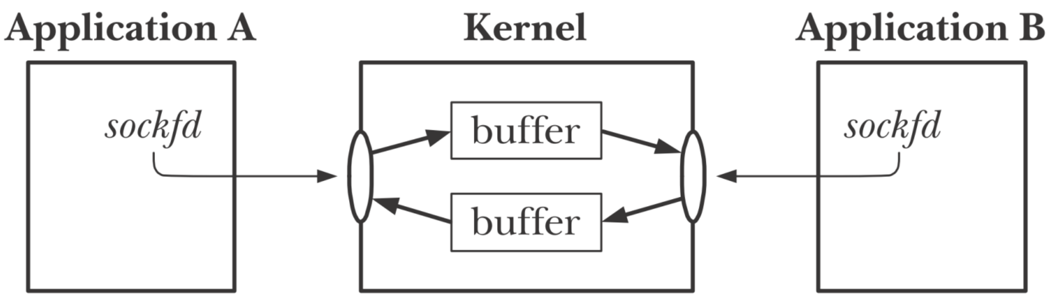
- TCP and UDP sockets are full duplex. File streams can also be full duplex.
- We can open a file stream on a socket using *fdopen()*.
- If we open with mode r+, then
  - An input function can’t be followed by output function without calling fseek().
  - An output function can’t followed by input function without calling fseek().
  - But we can’t call lseek() on sockets.
- So open two separate streams on a socket: one for reading, one for writing.

BITS Pilani
Pilani Campus

Unix Domain Sockets

Internet Domain vs Unix Domain

- Internet Domain: *AF_INET or AF_INET6*
  - Used in network communication.
  - Can be used between two processes in the same host also.
- Unix Domain: *AF_UNIX or AF_LOCAL*
  - Used for communication between processes on the same host. Same API as sockets API.
  - No TCP/IP protocol stack. A socket is made of two buffers in the kernel.
  - No header processing, no checksums. Reliable communication. Unix domain sockets are twice faster.



Unix Domain Sockets - Usage

- Unix domain sockets are used for three reasons:
  - Unix domain sockets are often twice as fast as a TCP socket when both peers are on the same host.
    - X Windows
  - used when passing file descriptors between processes on the same host.
  - unix domain sockets provide the client's process credentials (user ID and group IDs) to the server, which can provide additional security checking

Unix Domain Sockets

- Two types of sockets are provided.
  - Stream sockets
    - Similar to TCP
  - Datagram Sockets
    - Similar to UDP sockets.
    - Message boundaries are preserved.
    - Communication is reliable unlike UDP.

Unix Domain Socket Address

- End Point Address
  - pathnames within the normal file system
  - The pathname associated with a Unix domain socket should be an absolute pathname.

```
struct sockaddr_un {
    sa_family_t sun_family;/* Always AF_UNIX */
    char sun_path[108]; /* Null-terminated socket pathname */
};
```

Binding End Point to a Socket

- When used to bind a UNIX domain socket, bind() creates an entry in the file system.

```
const char *SOCKNAME = "/tmp/mysock";
int sfd;
struct sockaddr_un addr;
sfd = socket(AF_UNIX, SOCK_STREAM, 0);    /* Create socket */
if (sfd == -1)
    errExit("socket");
memset(&addr, 0, sizeof(struct sockaddr_un)); /* Clear structure */
addr.sun_family = AF_UNIX;    /* UNIX domain address */
strncpy(addr.sun_path, SOCKNAME, sizeof(addr.sun_path) - 1);
if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

  - We can't bind a socket to an existing pathname (bind() fails with the error EADDRINUSE).
  - A socket may be bound to only one pathname; conversely, a pathname can be bound to only one socket.
    - When the socket is no longer required, its pathname entry should be removed using unlink().

Unix Domain Stream Sockets

- Server
  - Absolute pathname is required.
  - Pathname specified in connect() should be existing, and bound to a socket
  - If the listening socket's queue is full, ECONREFUSED is immediately returned.
- Client
  - Create socket
  - Connect to the server.

Unix Domain Stream Server

```c
/*sockets/us_xfr_sv.c*/
main(int argc, char *argv[])
{   struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)  errExit("socket");
    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);
    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
    if (listen(sfd, BACKLOG) == -1)  errExit("listen");
    for (;;) {
        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)    errExit("accept");
        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
            if (write(STDOUT_FILENO, buf, numRead) != numRead)
                fatal("partial/failed write");
        if (numRead == -1) errExit("read");
        if (close(cfd) == -1) errMsg("close");
    }}
```

Unix Domain Stream Client

```
/*sockets/us_xfr_cl.c*/
main(int argc, char *argv[])
{   struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);  /* Create client socket */
    if (sfd == -1)  errExit("socket");
    /* Construct server address, and make the connection */
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);
    if (connect(sfd, (struct sockaddr *) &addr,
                sizeof(struct sockaddr_un)) == -1)
        errExit("connect");
    /* Copy stdin to socket */
    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            fatal("partial/failed write");
    if (numRead == -1)
        errExit("read");
    exit(EXIT_SUCCESS);  /* Closes our socket; server sees EOF */
}
```

Unix Domain Datagram Sockets

- Datagram sockets are reliable unlike UDP sockets.
  - Datagrams are not lost.
  - Datagrams are delivered in order and without duplicates.
- Server
  - Creates a socket
  - binds to well-known path.
- Client
  - Creates a socket
  - **binds the socket to an address, so that the server can send its reply.**
  - The client address is made unique by including the client's process ID in the pathname.

Unix Domain Datagram Server

```c
/*sockets/ud_ucase_sv.c*/
main(int argc, char *argv[])
{   struct sockaddr_un svaddr, claddr;
    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);       /* Create server socket */
    if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
        errExit("remove-%s", SV_SOCK_PATH);
    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);
    if (bind(sfd, (struct sockaddr *) &svaddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
    for (;;) {
        len = sizeof(struct sockaddr_un);
        numBytes = recvfrom(sfd, buf, BUF_SIZE, 0,
                            (struct sockaddr *) &claddr, &len);
        if (numBytes == -1)  errExit("recvfrom");
        printf("Server received %ld bytes from %s\n", (long) numBytes,
                claddr.sun_path);
        for (j = 0; j < numBytes; j++)
            buf[j] = toupper((unsigned char) buf[j]);
        if (sendto(sfd, buf, numBytes, 0, (struct sockaddr *) &claddr, len) !=
                numBytes)
            fatal("sendto");
    }}
```

Unix Domain Datagram Client

```
/* sockets/ud_ucase_cl.c*/
main(int argc, char *argv[])
{   struct sockaddr_un svaddr, claddr;
     sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    memset(&claddr, 0, sizeof(struct sockaddr_un));
    claddr.sun_family = AF_UNIX;
    snprintf(claddr.sun_path, sizeof(claddr.sun_path),
            "/tmp/ud_ucase_cl.%ld", (long) getpid());
if (bind(sfd, (struct sockaddr *) &claddr, sizeof(struct sockaddr_un)) == -1)
        errExit("bind");
    /* Construct address of server */
    memset(&svaddr, 0, sizeof(struct sockaddr_un));
    svaddr.sun_family = AF_UNIX;
    strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(svaddr.sun_path) - 1);
    /* Send messages to server; echo responses on stdout */
    for (j = 1; j < argc; j++) {
        msgLen = strlen(argv[j]);       /* May be longer than BUF_SIZE */
        if (sendto(sfd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
                sizeof(struct sockaddr_un)) != msgLen)
            fatal("sendto");
        numBytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL, NULL);
        if (numBytes == -1)   errExit("recvfrom");
        printf("Response %d: %.*s\n", j, (int) numBytes, resp); }
    remove(claddr.sun_path);/* Remove client socket pathname */
}
```

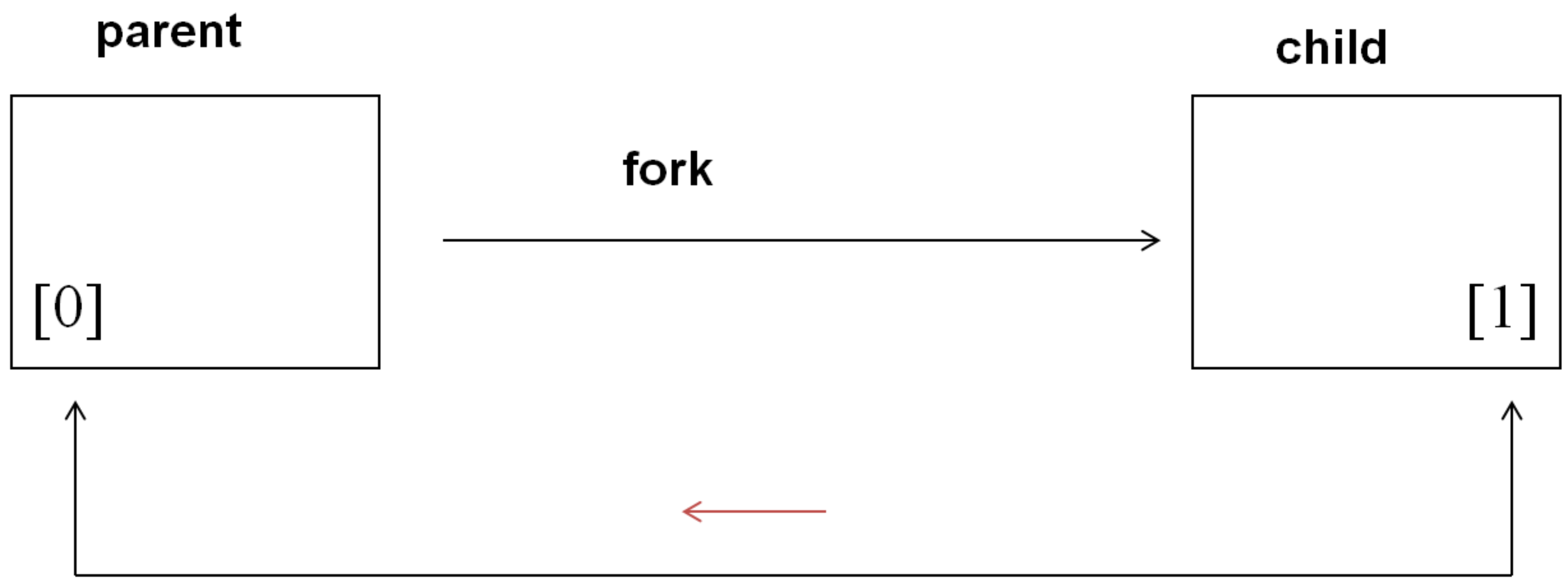
socketpair()

- creates a pair of sockets and connect them together.

```
#include <sys/socket.h>
int socketpair(int  domain, int  type ,int  protocol ,int  sockfd [2]);
//Returns 0 on success, or -1 on error
```

  - Returns two socket fds.
  - No path names bound for sockets. Not visible outside the process.
  - Type SOCK_STREAM creates the equivalent of a bidirectional pipe (also known as a stream pipe).
  - Each socket can be used for both reading and writing.
  - Just like in pipe, after creating calling socketpair(), fork() is called.
  - Parent and child can communicate using sockets.

socketpair()



socketpair()

```c
#include <sys/types.h>
#include <unistd.h>
#include <sys/socket.h>
main ()
{
  int i;
  int p[2];
  pid_t ret;
    socketpair(AF_UNIX, SOCK_STREAM, 0, p);
  ret = fork ();
  if (ret == 0)
    {
      close (1);
      dup (p[1]);
      close (p[0]);
      execlp ("ls", "ls", "-l", (char *) 0);
    }
  if (ret > 0)
    {
      close (0);
      dup (p[0]);
      close (p[1]);
  execlp ("wc", "wc", "-l", (char *) 0);
    }
}
```

Passing File Descriptors

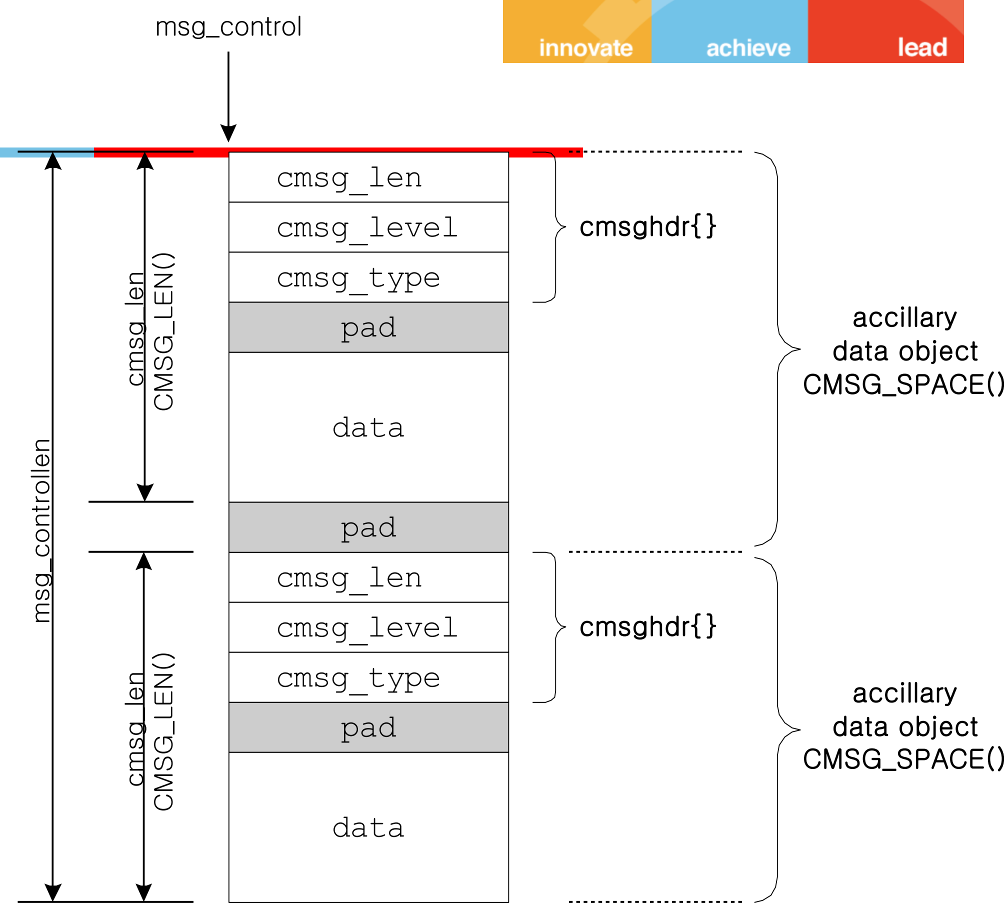
- Unix system provide a way to pass any open descriptor from one process to any other process.(using *sendmsg()*)
- It allows one process (typically a server) to do the privileged execution
  - dialing a modem, negotiating locks for the file or deal with database
  - simply pass back to the calling process a descriptor that can be used with all the I/O functions.
- All the details involved in opening the file or device are hidden from the client.

Passing File Descriptors

- Steps involved
  - Create a unix domain socket(stream or datagram)
  - one process opens a descriptor by calling any of the unix function that returns a descriptor
  - the sending process build a ***msghdr*** structure containing the descriptor to be passed
  - Sending process sends ancillary data using ***sendmsg()*** with SCM_RIGHTS
  - the receiving process calls ***recvmsg()*** to receive the descriptor on the unix domain socket
- Passing a descriptor is not same as passing descriptor number
  - involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel.

Ancillary Data

- Ancillary data can be sent and received using the msg_control and msg_controllen members of the msghdr structure.
  - Another term for ancillary data is control information.




```c
struct cmsghdr {
  socklen_t  cmsg_len;   /* length in bytes, including this structure */
  int        cmsg_level; /* originating protocol */
  int        cmsg_type;  /* protocol-specific type */
      /* followed by unsigned char cmsg_data[] */
};
```

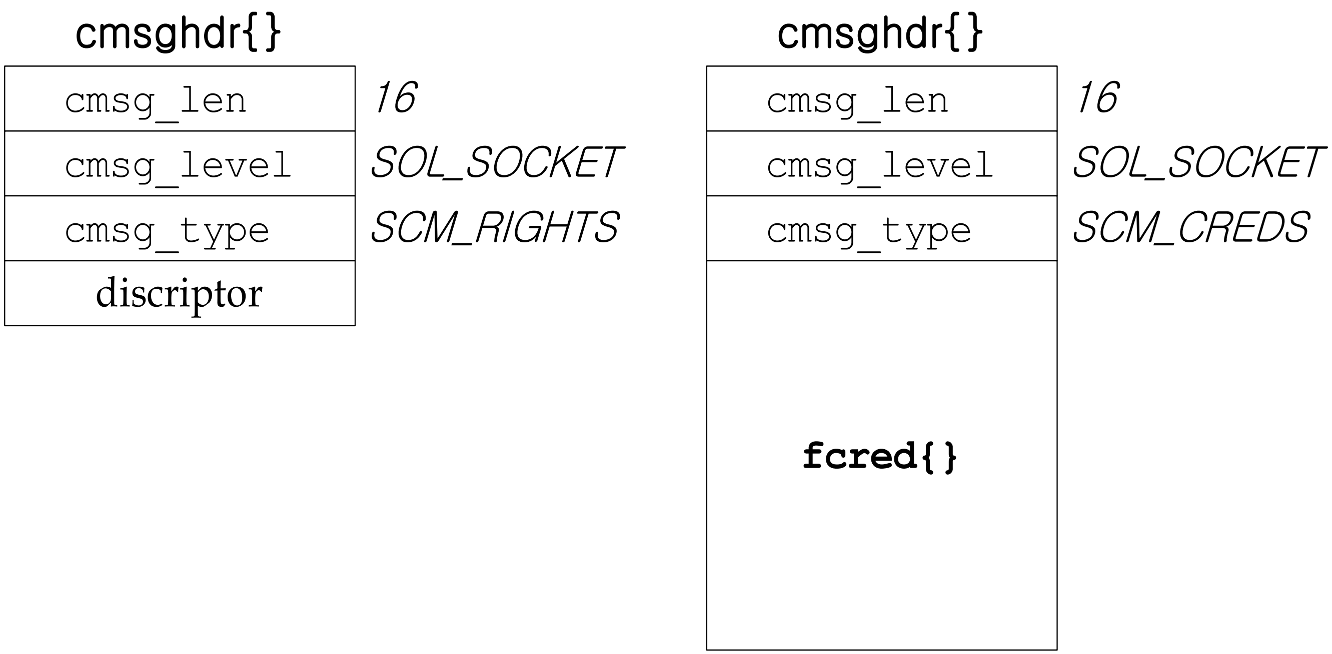
Ancillary Data

- Ancillary data is domain specific.

Protocol	cmsg_level	Cmsg_type	Description
IPv4	IPPROTO_IP	IP_RECVDSTADDR IP_RECVIF	receive destination address with UDP datagram receive interface index with UDP datagram
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS IPV6_HOPLIMIT IPV6_HOPOPTS IPV6_NEXTHOP IPV6_PKTINFO IPV6_RTHDR	specify / receive destination options specify / receive hop limit specify / receive hop-by-hop options specify next-hop address specify / receive packet information specify / receive routing header
Unix domain	SOL_SOCKET	SCM_RIGHTS SCM_CREDS	send / receive descriptors send / receive user credentials

Ancillary Data

- File descriptors and process credentials can be passed between unrelated processes using ancillary data.



Passing File Descriptors Example

- Protocol
  - If buf[1]<0 then there is error.
  - If buf[1]=0 then it is success.
- Receiver
  - Create unix domain stream socket
  - Send file name
  - Call recv_fd
- Sender
  - Create unix domain stream socket
  - Open the file descriptor
  - Call send_fd

Macros

- Macros associated with ancillary data

```
#include <sys/socket.h>
#include <sys/param.h> /* for ALIGN macro on many implementations */
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mhdrptr) ;
//Returns: pointer to first cmsghdr structure or NULL if no ancillary data
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mhdrptr, struct cmsghdr *cmsgptr) ;
//Returns: pointer to next cmsghdr structure or NULL if no more ancillary data
unsigned char *CMSG_DATA(struct cmsghdr *cmsgptr) ;
//Returns: pointer to first byte of data associated with cmsghdr structure
unsigned int CMSG_LEN(unsigned int length) ;
//Returns: value to store in cmsg_len given the amount of data
unsigned int CMSG_SPACE(unsigned int length) ;
//Returns: total size of an ancillary data object given the amount of data
```

Send fd

```c
#include <sys/socket.h>
#define CONTROLLEN  CMSG_LEN(sizeof(int))
static struct cmsghdr   *cmptr = NULL;
int send_fd(int fd, int fd_to_send)
{   struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */
    iov[0].iov_base = buf; iov[0].iov_len  = 2;
    msg.msg_iov     = iov; msg.msg_iovlen  = 1;
    msg.msg_name    = NULL; msg.msg_namelen = 0;
if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
}
else { if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send;     /* the fd to pass */
        buf[1] = 0;             /* zero status means OK */
    }
    buf[0] = 0;                 /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Receive fd

```c
int recv_fd(int sockfd )
{
    #define CONTROLLEN  CMSG_LEN(sizeof(int))
    static struct cmsghdr   *cmptr = NULL;

    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */
    memset(&msg, 0, sizeof(msg));
    iov.iov_base   = buf;
    iov.iov_len    = sizeof(data)-1;
    msg.msg_iov    = &iov;
    msg.msg_iovlen = 1;
    if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
       return(-1);
    msg.msg_control = cmptr;
    msg.msg_controllen = CONTROLLEN;
    recvmsg(sockfd, &msg, 0)
        if (buf[1]<0)) {
                printf("failed to open %s: %s\n", name, data);
                return -1;
        }
        /* Loop over all control messages */
        cmsg = CMSG_FIRSTHDR(&msg);
        while (cmsg != NULL) {
                if (cmsg->cmsg_level == SOL_SOCKET
                 && cmsg->cmsg_type  == SCM_RIGHTS)
                        return *(int *) CMSG_DATA(cmsg);
                cmsg = CMSG_NXTHDR(&msg, cmsg);
        }
}
```

Passing Credentials

innovate achieve lead

- A process can pass its credentials as ancillary data using SCM_CREDS option.
- The structure for credentials

```
struct ucred {
    pid_t pid;    /* process ID of the sending process */
    uid_t uid;    /* user ID of the sending process */
    gid_t gid;    /* group ID of the sending process */
};
```

- This structure is filled by the kernel and passed onto the receiver process.
- Example:
    - Sender process sends file name and access mode.
    - Server verifies the credentials and passes on the fd.

Sender

```
sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path, "PATH");
connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
msgh.msg_iov = &iov;
msgh.msg_iovlen = 1;
/* Send Filename and Access Mode to server */
strcat(data, argv[1]);strcat(data, "#");
strcat(data, argv[2]);strcat(data, "#");
iov.iov_base = data;
iov.iov_len = MAX_DATA;
msgh.msg_name = NULL;
msgh.msg_namelen = 0;
msgh.msg_control = NULL;
msgh.msg_controllen = 0;

if(sendmsg(sockfd, &msgh, 0) < 0)
{
    perror("Error sending message");
    exit(1);
}
```

Receiver

```
optval = 1;
/* Set SO_PASSCRED socket option for receving credentials of other processes */
setsockopt(*(int *)arg, SOL_SOCKET, SO_PASSCRED, &optval, sizeof(optval));
/* Set 'control_un' to describe ancillary data that we want to receive */
control_un.cmh.cmsg_len = CMSG_LEN(sizeof(struct ucred));
control_un.cmh.cmsg_level = SOL_SOCKET;
control_un.cmh.cmsg_type = SCM_CREDENTIALS;
/* Set 'msgh' fields to describe 'control_un' */
msgh.msg_control = control_un.control;
msgh.msg_controllen = sizeof(control_un.control);
msgh.msg_iov = &iov;    msgh.msg_iovlen = 1;
iov.iov_base = data;
iov.iov_len = MAX_DATA;
msgh.msg_name = NULL;
msgh.msg_namelen = 0;
/* Receive real plus ancillary data */
nr = recvmsg(*(int *)arg, &msgh, 0);
/* Extract credentials information from received ancillary data */
cmhp = CMSG_FIRSTHDR(&msgh);
ucredp = (struct ucred *) CMSG_DATA(cmhp);
printf("Received Credentials pid: %ld, uid: %ld, gid: %ld\n",
(long) ucredp->pid, (long) ucredp->uid, (long) ucredp->gid);
```

The Linux Abstract Socket Namespace

- Is a Linux-specific feature that allows us to bind a UNIX domain socket to a name without that name being created in the file system.
- It is not necessary to unlink the socket pathname when we have finished using the socket. The abstract name is automatically removed when the socket is closed.
  - To create an abstract binding, we specify the first byte of the sun_path field as a null byte (\0).
  - The remaining bytes of the sun_path field then define the abstract name for the socket. These bytes are interpreted in their entirety, rather than as a null-terminated string.

The Linux Abstract Socket Namespace Example

```
struct sockaddr_un addr;
memset(&addr, 0, sizeof(struct sockaddr_un));  /* Clear address structure */
addr.sun_family = AF_UNIX;                     /* UNIX domain address */
/* addr.sun_path[0] has already been set to 0 by memset() */
strncpy(&addr.sun_path[1], "xyz", sizeof(addr.sun_path) - 2);
            /* Abstract name is "xyz" followed by null bytes */
sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sockfd == -1)
    errExit("socket");
if (bind(sockfd, (struct sockaddr *) &addr,
        sizeof(struct sockaddr_un)) == -1)
    errExit("bind");
```

innovate achieve lead

BITS Pilani
Pilani Campus

Unix I/O Models

IO Models

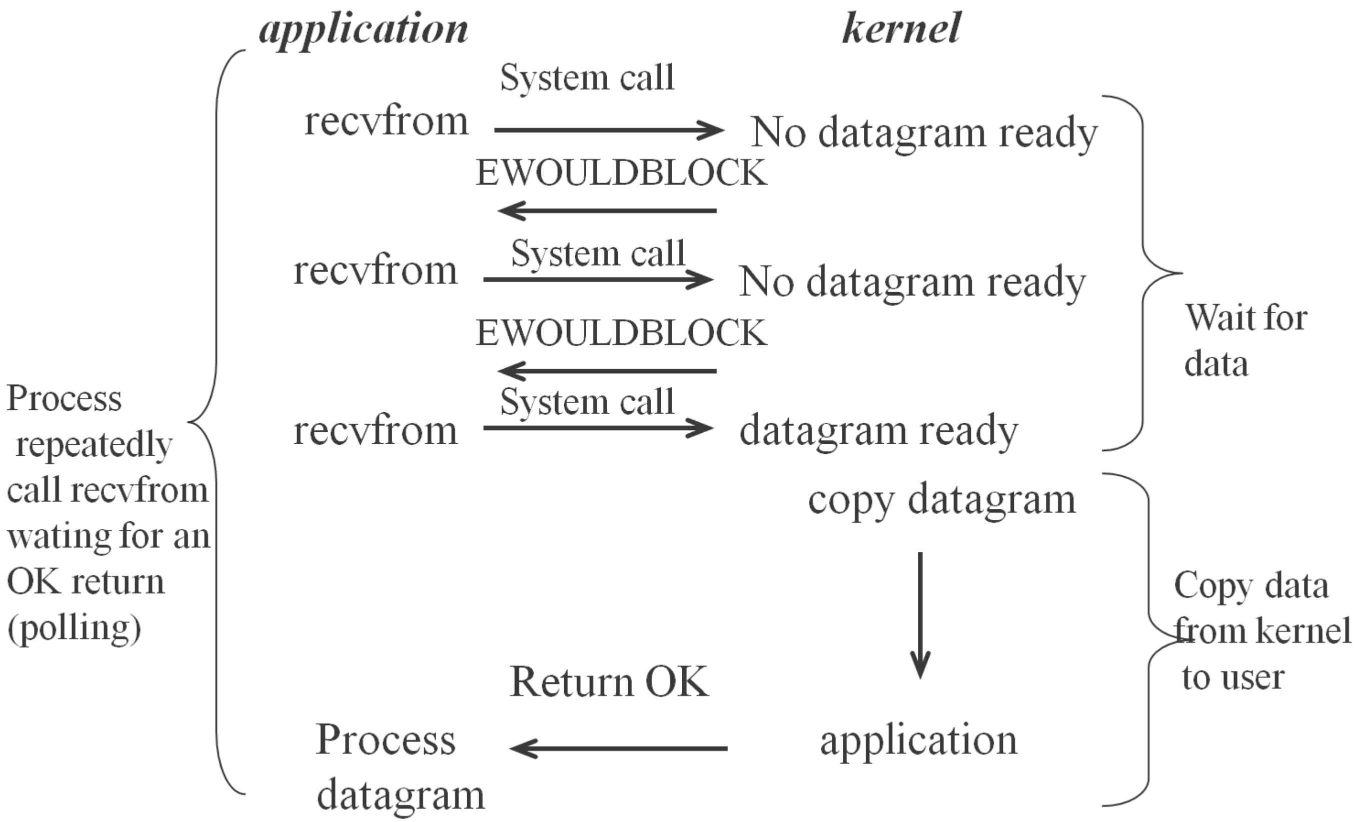
- While doing I/O there are two phases
  - Waiting for the data
  - Copying the data
- Each I/O model differs how it deals with these two phases.
- There are five I/O models
  - blocking I/O
  - nonblocking I/O
  - I/O multiplexing (select and poll)
  - signal driven I/O (SIGIO)
  - asynchronous I/O (the POSIX aio_functions)

Blocking I/O Model

- Most prevalent model

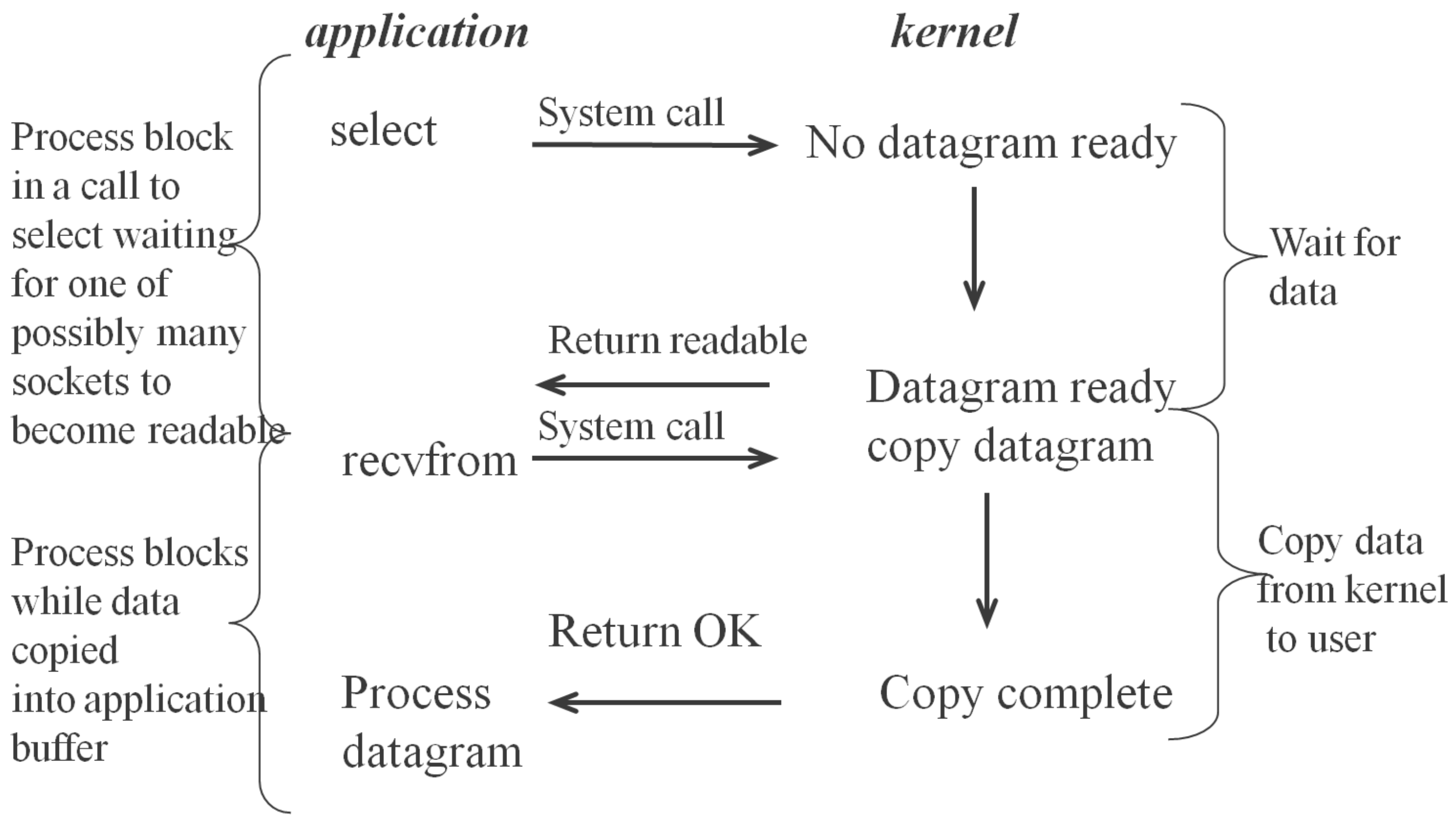
Nonblocking I/O Model

- When the socket is set to be non-blocking,
  - We tell the kernel that do not put the process to sleep if IO can’t be completed.



I/O Multiplexing Model

- Block in select() or poll() instead of blocking in actual I/O system call.

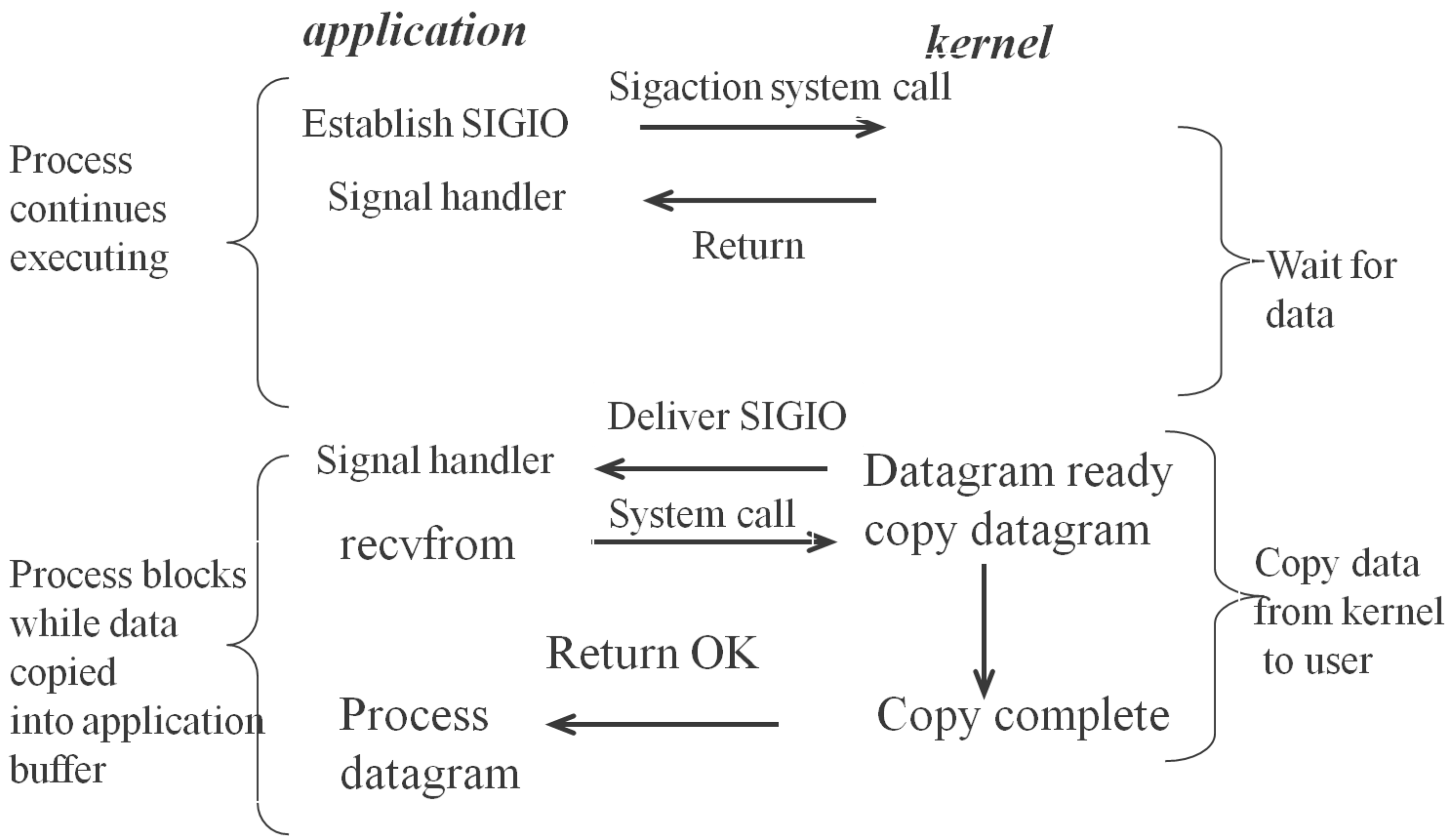


I/O Multiplexing Model

- Blocking and I/O multiplexing seem to be non-different and actually calling two sys calls in I/O multiplexing.
- Advantage with I/O multiplexing is that it can wait for I/O on multiple fds.

Signal-Driven I/O Model

- Tell the kernel to notify us with the SIGIO signal when the descriptor is ready.

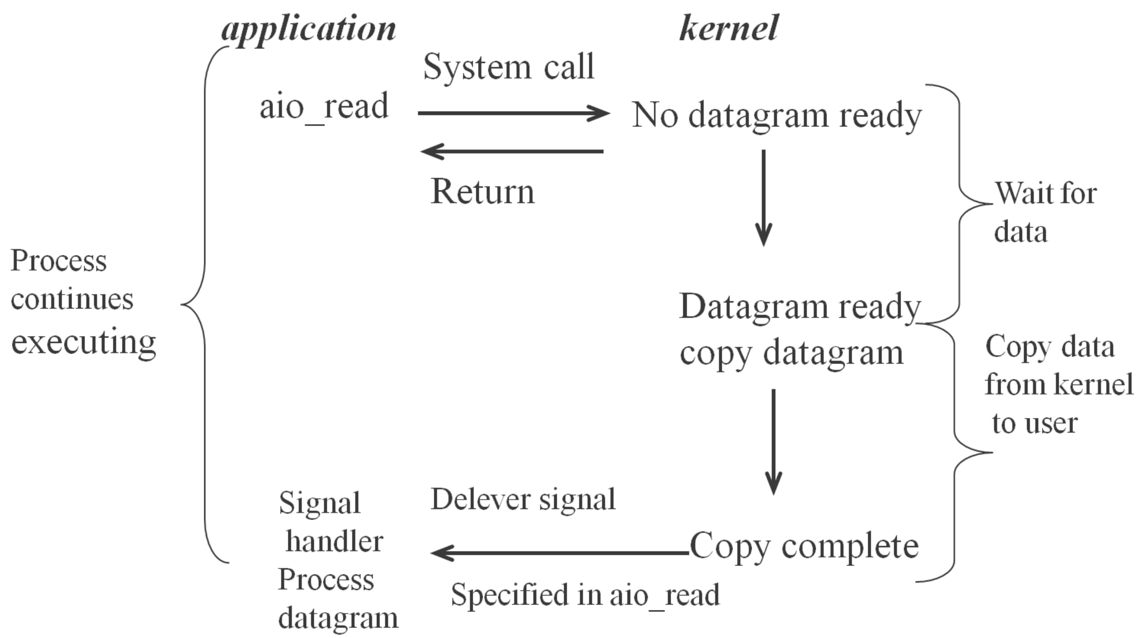


Signal-Driven I/O Model

- To use signal-driven I/O with a socket (SIGIO) requires the process to perform the following three steps:
  - A signal handler must be established for the SIGIO signal.
  - The socket owner must be set, normally with the F_SETOWN command of fcntl.
  - Signal-driven I/O must be enabled for the socket, normally with the F_SETFL command of fcntl to turn on the O_ASYNC flag.

Asynchronous I/O Model

- The main difference between this model and the signal-driven I/O models that
  - with signal-driven I/O, the kernel tells us when an I/O operation can be initiated,
  - but with asynchronous I/O, the kernel tells us when an I/O operation is complete.



Asynchronous I/O Model

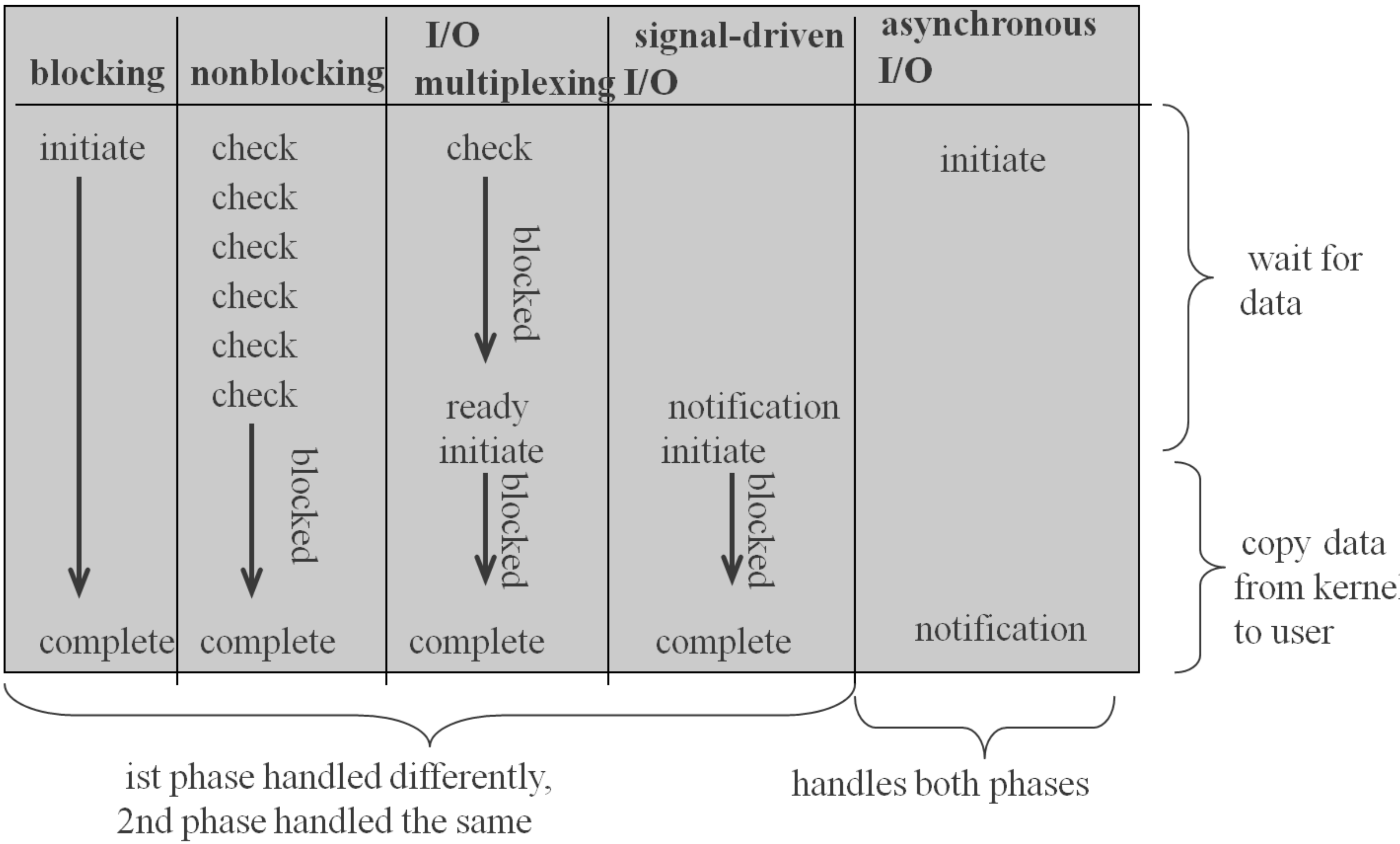
- POSIX API for asynchronous IO is implemented in a very few systems.
- *aiocb* structure

```
struct aiocb{
int               aio_fildes //     file descriptor
off_t             aio_offset //     file offset
volatile void*    aio_buf //        location of buffer
size_t            aio_nbytes //     length of transfer
int               aio_reqprio //    request priority offset
struct sigevent aio_sigevent   // signal number and value
int               aio_lio_opcode //operation to be performed
};
```

```
#include <aio.h>
int aio_read(struct aiocb *aiocbp);
```

Comparison of I/O Models

- First four models differ only in first phase.



Synchronous I/O vs Asynchronous I/O

- POSIX defines these two terms as follows:
  - A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
  - An asynchronous I/O operation does not cause the requesting process to be blocked.
- Using these definitions,
  - the first four I/O models
    - blocking,
    - nonblocking,
    - I/O multiplexing,
    - and signal-driven I/O
  - are all synchronous because the actual I/O operation (recvfrom) blocks the process.
  - Only the asynchronous I/O model matches the asynchronous I/O definition.

innovate achieve lead

BITS Pilani
Pilani Campus

I/O Multiplexing

I/O Multiplexing

- I/O multiplexing allows us to simultaneously monitor multiple file descriptors to see if I/O is possible on any of them.
- select(), appeared along with the sockets API in BSD. This was historically the more widespread of the two system calls. The other system call, poll(), appeared in System V.
- We can use select() and poll() to monitor file descriptors for regular files, terminals, pseudoterminals, pipes, FIFOs, sockets, and some types of character devices.
- Both system calls allow a process either to block indefinitely waiting for file descriptors to become ready or to specify a timeout on the call.

select()

- The select() system call blocks until one or more of a set of file descriptors becomes ready.

```c
#include <sys/time.h>   /* For portability */
#include <sys/select.h>
int select(int  nfds , fd_set * readfds , fd_set * writefds,
            fd_set * exceptfds, struct timeval * timeout );
//Returns number of ready file descriptors, 0 on timeout, or -1 on error
```

  - *nfds*: highest number assigned to a descriptor +1.
  - *readfds*: set of descriptors we want to read from.
  - *writefds*: set of descriptors we want to write to.
  - *exceptfds*: set of descriptors to watch for exceptions.
  - *timeout*: maximum time select should wait

```c
struct timeval {
    long tv_usec;     /* seconds */
    long tv_usec;   /* microseconds */
}
```

select()

- timeval==NULL
  - Wait forever : return only when descriptor is ready
- timeval != NULL: wait up to a fixed amount of time
  - timeval = 0
    - Do not wait at all : return immediately after checking the descriptors
  - Timeval>0
    - Return only if descriptor is ready or timeval expires.

File descriptor sets

- The readfds, writefds, and exceptfds arguments are pointers to file descriptor sets, represented using the data type *fd_set*.
- the fd_set data type is implemented as a bit mask.

```
#include <sys/select.h>
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 bit for fd on in fdset ? */
 //Returns true (1) if fd is in fdset, or false (0) otherwise
```

- A file descriptor set has a maximum size, defined by the constant FD_SETSIZE . On Linux, this constant has the value 1024.

select()

- Readset
  - descriptor for checking readable
- Writeset
  - descriptor for checking writable
- exceptset
  - descriptor for checking two exception conditions
    - arrival of out of band data for a socket
    - he presence of control status information to be read from the master side of a pseudo terminal
- When select returns value>1, these sets have been modified by kernel. Now they contain the fds which are ready.

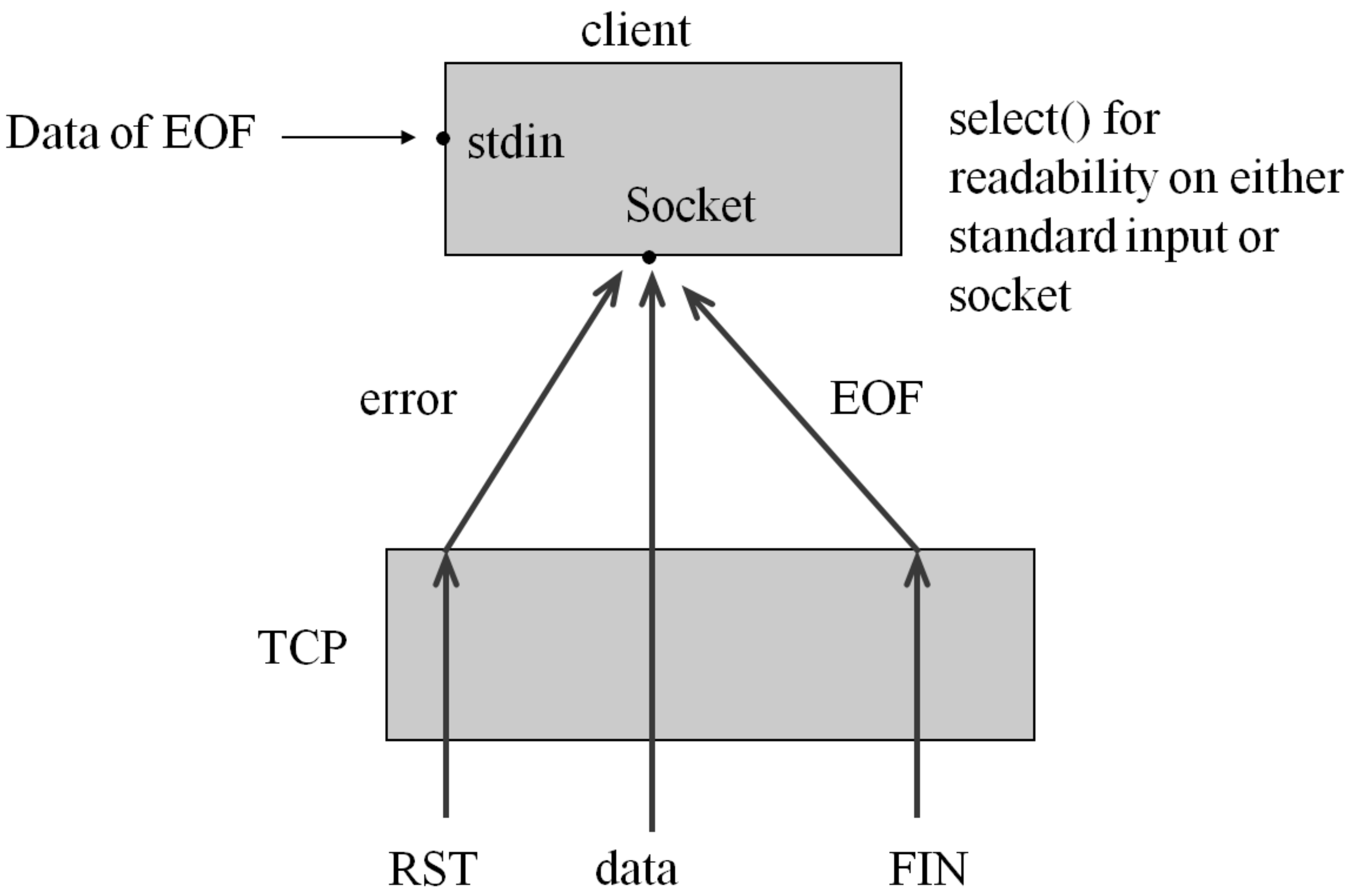
When is the descriptor ready for reading?

- The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer. SO_RCVLOWAT socket option. It defaults to 1 for TCP and UDP sockets
- The read half of the connection is closed (i.e., a TCP connection that has received a FIN)
- The socket is a listening socket and the number of completed connections is nonzero.
- A socket error is pending. A read operation on the socket will not block and will return an error (–1) with errno set to the specific error condition.

When the socket is ready for writing?

- The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer. 2048 bytes.
- The write half of the connection is closed. A write operation on the socket will generate SIGPIPE.
- A socket using a non-blocking connect has completed the connection, or the connect has failed
- A socket error is pending. A write operation on the socket will not block and will return an error (–1) with errno set to the specific error condition.
- These pending errors can also be fetched and cleared by calling getsockopt with the SO_ERROR socket option.

Condition handled by select in a client



Three conditions are handled with the socket

- Peer TCP send a data,the socket becomr readable and *read* returns greater than 0
- Peer TCP send a FIN(peer process terminates), the socket become readable and *read* returns 0(end-of-file)
- Peer TCP send a RST(peer host has crashed and rebooted), the socket become readable and returns -1 and *errno* contains the specific error code

```
void  str_cli(FILE *fp, int sockfd)
{
int     maxfdp1;
fd_set  rset;
char    sendline[MAXLINE], recvline[MAXLINE];
FD_ZERO(&rset);
for ( ; ; ) {
    FD_SET(fileno(fp), &rset);
    FD_SET(sockfd, &rset);
    maxfdp1 = max(fileno(fp), sockfd) + 1;
    select(maxfdp1, &rset, NULL, NULL, NULL);
    if (FD_ISSET(sockfd, &rset)) {    /* socket is readable */
    if (Readline(sockfd, recvline, MAXLINE) == 0)
    err_quit("str_cli: server terminated prematurely");
    Fputs(recvline, stdout);
    }
    if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
    if (Fgets(sendline, MAXLINE, fp) == NULL)
    return;      /* all done */
    Writen(sockfd, sendline, strlen(sendline));
    }
}//for
}//str_cli
```

Acknowledgements

Q&A

Thank You