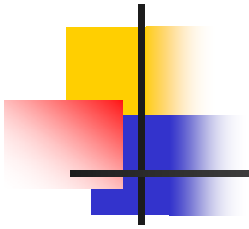
Computer Network Programming

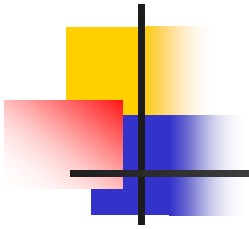
I/O Multiplexing

Dr. Sam Hsu
Computer Science & Engineering
Florida Atlantic University



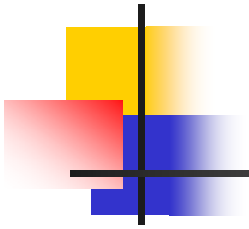
I/O Multiplexing

- I/O Models
- The select() Function
- The timeval Structure
- Ready Conditions for Descriptors
- Low Water Mark
- Stop-n-Wait vs. Batch Mode Operations
- The shutdonw() Function
- Client/Server Revisited
- The pselect() Function
- Server Designs



A Scenario

- Given the echo client introduced earlier (*str_cli.c*) , what if it is blocked in a call to Fgets(), and the echo server is terminated?
  - The server TCP correctly sends a FIN to the client TCP, but the client process never sees it until the client process reads from the socket later.
- What we need here is the capability to handle multiple I/O descriptors at the same time.
  - I/O multiplexing!



I/O Models

- Five UNIX I/O models
  - Blocking I/O
  - Nonblocking I/O
  - I/O multiplexing
  - Signal driven I/O
  - Asynchronous I/O
- Note: An input operation typically involves two distinct phases:
  - Waiting for data to be ready.
  - Copying data from kernel to process.

Blocking I/O Model

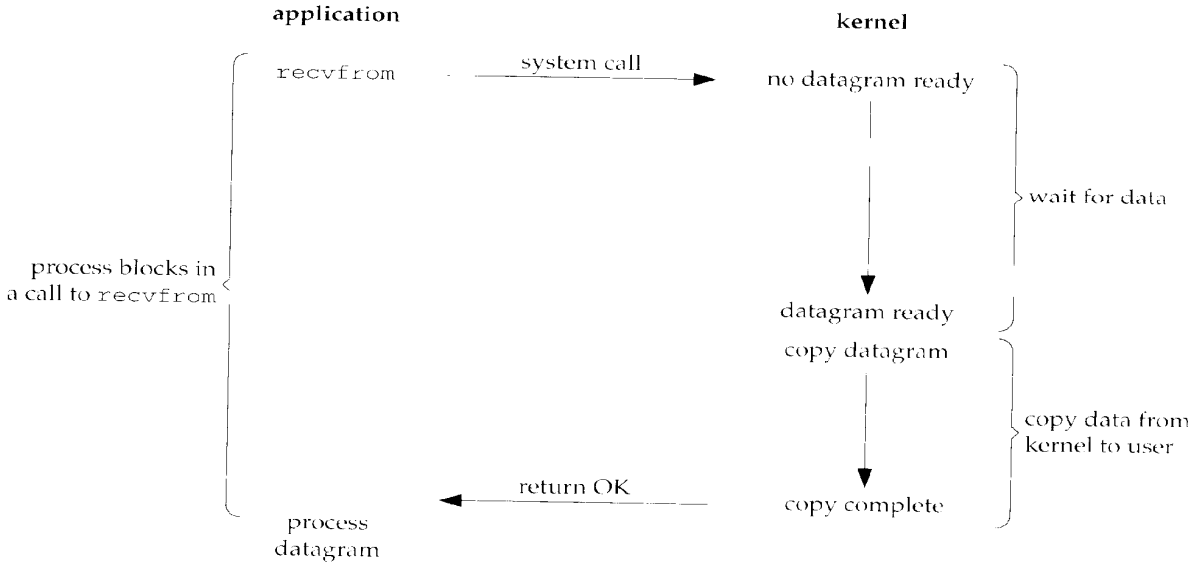


Figure 6.1 Blocking I/O model.

- A process that performs an I/O operation will wait (block) until the operation is completed.
- By default, all sockets I/Os are blocking I/Os.

Nonblocking I/O Model

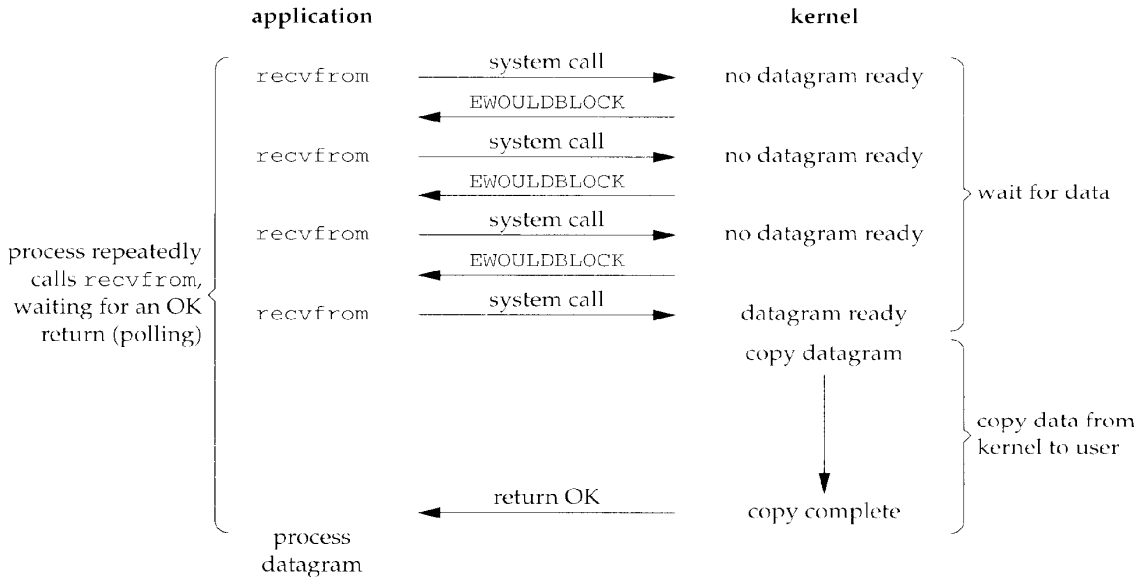


Figure 6.2 Nonblocking I/O model.

- When a process cannot complete an I/O operation, instead of putting the process to sleep, the kernel will return an error (EWOULDBLOCK) to the process to indicate the requested I/O operation not completed.

I/O Multiplexing Model

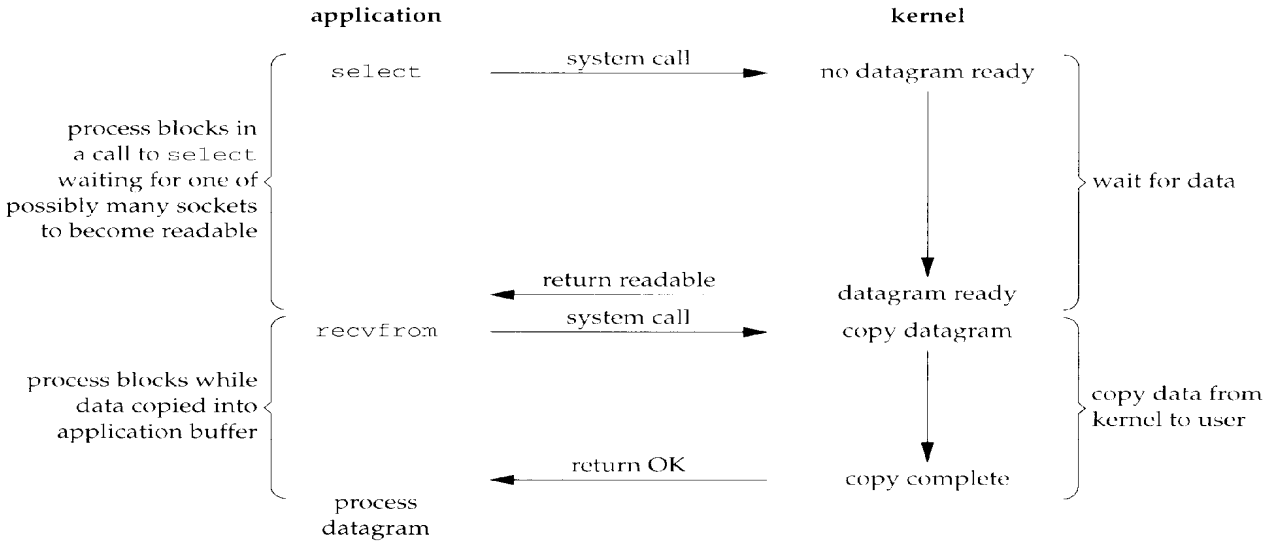


Figure 6.3 I/O multiplexing model.

- For dealing with multiple I/O resources concurrently.
- Implemented using select()/poll().
- Used before actual I/O systems calls.
- Are blocking operations.

Signal Driven I/O Model

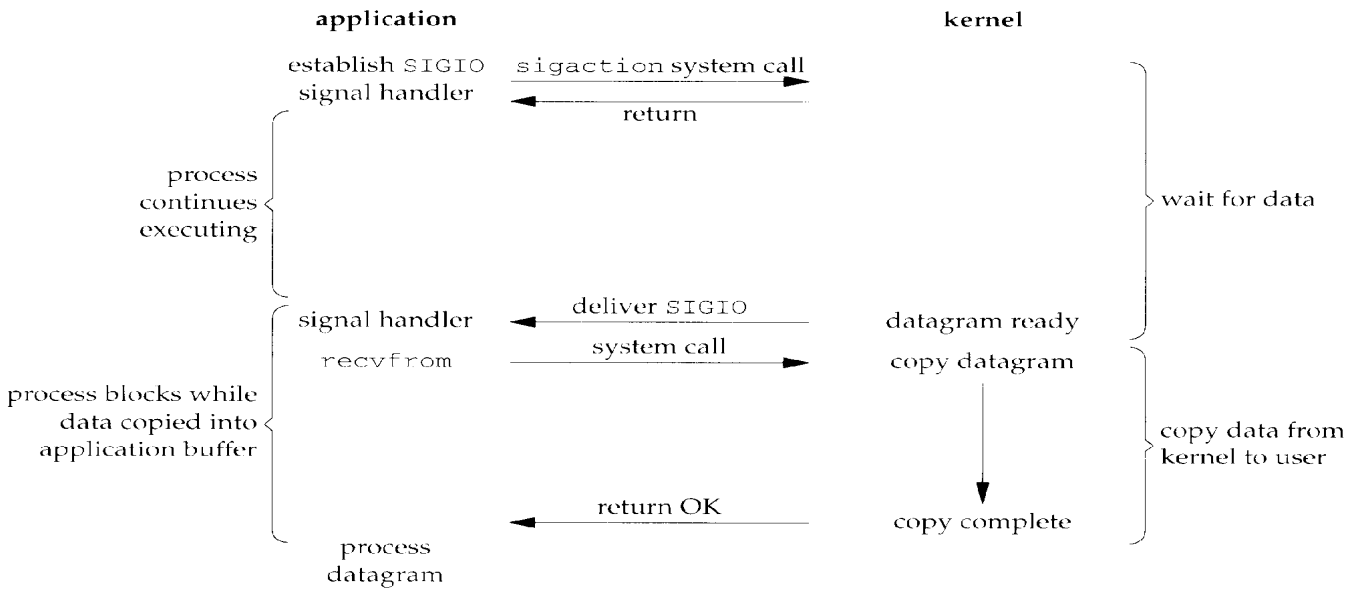


Figure 6.4 Signal Driven I/O model.

- A process is notified by the kernel via the SIGIO signal when a requested I/O resource is ready.
- In need of establishing a signal handler for SIGIO.

Asynchronous I/O Model

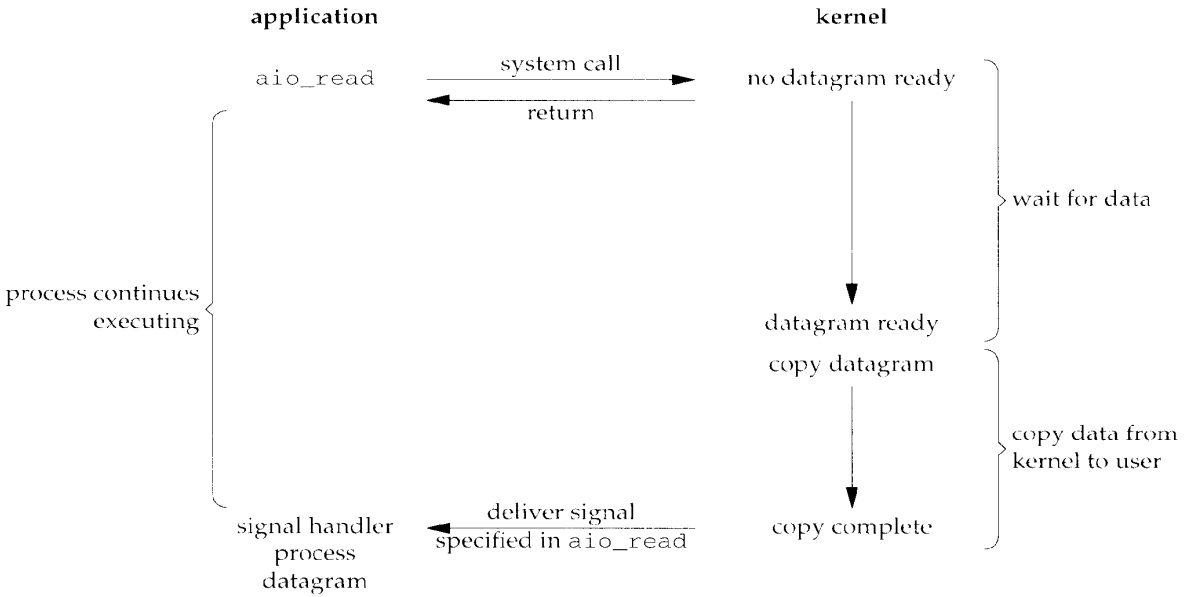


Figure 6.5 Asynchronous I/O model.

- A process is notified by the kernel via a preset signal when a requested I/O operation is complete.
- Use aio_read()/aio_write(), including a signal for notification.

Comparison of I/O Models

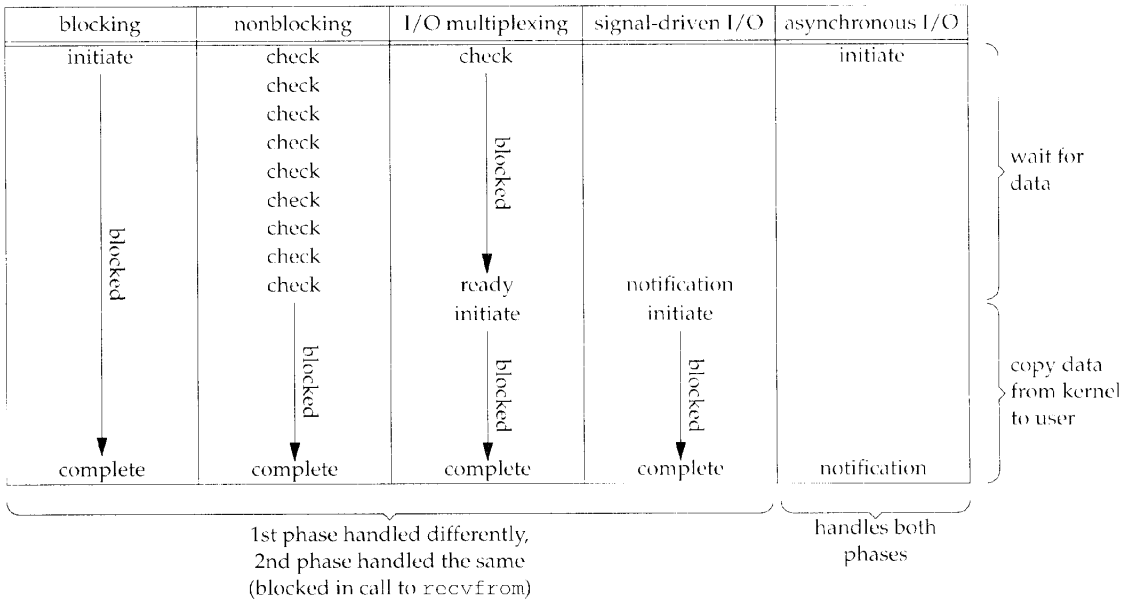
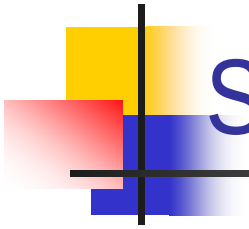


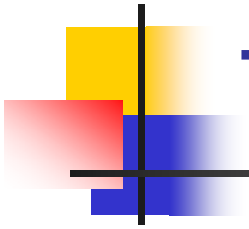
Figure 6.6 Comparison of the five I/O models.

- Note: A synchronous I/O operation blocks the requesting process, whereas an asynchronous I/O operation does not block the requesting process, while waiting for the requested I/O operation to complete.



Synchronous versus Asynchronous

- According to POSIX definitions:
  - 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.



The *select()* Function (1/2)

- Is used to tell the kernel to notify the calling process when some event(s) of interest occurs.
  - For example,
    - Any descriptor in the set { 1, 2, 4} is ready for reading.
    - Any descriptor in the set { 3, 5} is ready for writing.
    - Any descriptor in the set { 2, 3, 6} has an exception pending.
    - After waiting for 5 seconds and 40 milliseconds.

The *select()* Function (2/2)

- Syntax:

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

int select(int maxfdp1, fd_set *readset, fd_set *writeset,
           fd_set *exceptset, const struct timeval *timeout);

   Returns: positive count of ready descriptors, 0 on timeout, -1 on error
```

  - *Maxfdp1*: max descriptor value plus 1
    (total number of descriptors in all sets)
    (hint: descriptor values: 0, 1, 2, **…**)

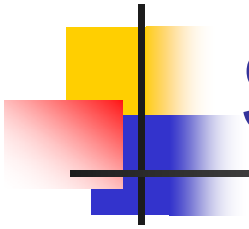
The timeval Structure

- Syntax:

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

- Three uses for select():
  - Wait forever if the structure pointer is NULL.
  - Wait up to tv_sec and tv_usec.
  - No wait if both tv_sec and tv_usec set to 0.

Note: The actual timeout value resolution is up to implementation. Some Unix kernel rounds the timeout value up to a multiple of 10 ms. There may also very likely be a scheduling latency involved.



Supporting Functions

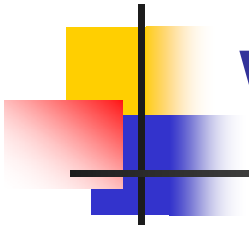
- 4 functions defined:

  void FD_ZERO(fd_set **fdset*);
  - To clear all bits in *fdset.*

  void FD_SET(int *fd,* fd_set **fdset*);
  - To turn on the bit for *fd* in *fdset.*

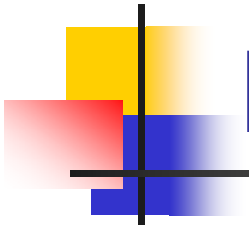
  void FD_CLR(int *fd*, fd_set **fdset*);
  - To turn off the bit for *fd* in *fdset.*

  int FD_ISSET(int *fd*, fd_set **fdset*);
  - To test to see if the bit for *fd* is on in *fdset.*



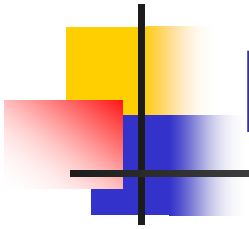
When Is A Descriptor Ready?

- Ready for read.
  - A read operation will not block.
- Ready for write.
  - A write operation will not block.
- Presence of exception data.
  - E.g., out-of-band data received, some control status information from a master pseudo-terminal, etc.



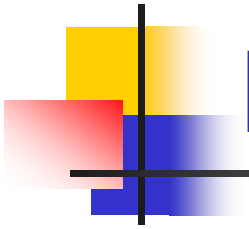
Ready for Read

- Data received is >= read buffer *low-water* mark.
  - Can be set using SO_RCVLOWAT.
  - Default is 1 for TCP and UDP sockets.
- The read-half of the connection is closed.
  - A read operation will return 0 (EOF) without blocking.
- A listening socket with an established connection.
  - An accept operation on the socket will normally not block.
- A socket error is pending.
  - A read operation on the socket will return an error (-1) without blocking.



Ready for Write

- Available space in send buffer is >= *low-water* mark.
  - Can be set using SO_SNDLOWAT.
  - Default is 2048 for TCP and UDP sockets.
- The write-half of the connection is closed.
  - A write operation will generate SIGPIPE without blocking.
- A socket using a non-blocking connect() has completed the connection, or the connect() call has failed.
- A socket error is pending.
  - A write operation on the socket will return an error (-1) without blocking.

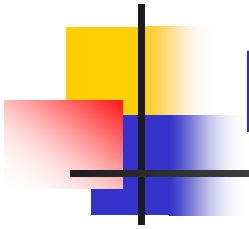


Low-Water Mark

- The purpose of read/write *low-water marks* is to give the application control over how much data must be available for reading/writing before select() returns readable or writable.
- Note: when a descriptor is writable, SO_SNDLOWAT indicates the minimum available write buffer size. One may not know how much of the buffer is actually available to be filled.
  - The same holds for a read descriptor.

Ready Conditions for select()

Condition	Readable?	Writable?	Exception?
Data to read	•		
Read-half connection closed	•		
New connection ready for listening socket	•		
Space available for writing		•	
Write-half connection closed		•	
Pending error	•	•	
TCP out-of-band data			•



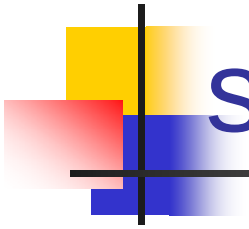
Process Alternative

- The use of blocking I/O and select() can achieve the similar behavior/effect of nonblocking I/O. However, there is another alternative — using processes.
  - One may fork() processes and have each process handle only one direction of I/O.
    - E.g., Process 1 reads from stdin (blocking) to network, and Process 2 reads from network (blocking) to stdout.
    - Beware of process synchronization issues.

str_cli() Using select() (1/2)

- *select/strcliselect01.c*

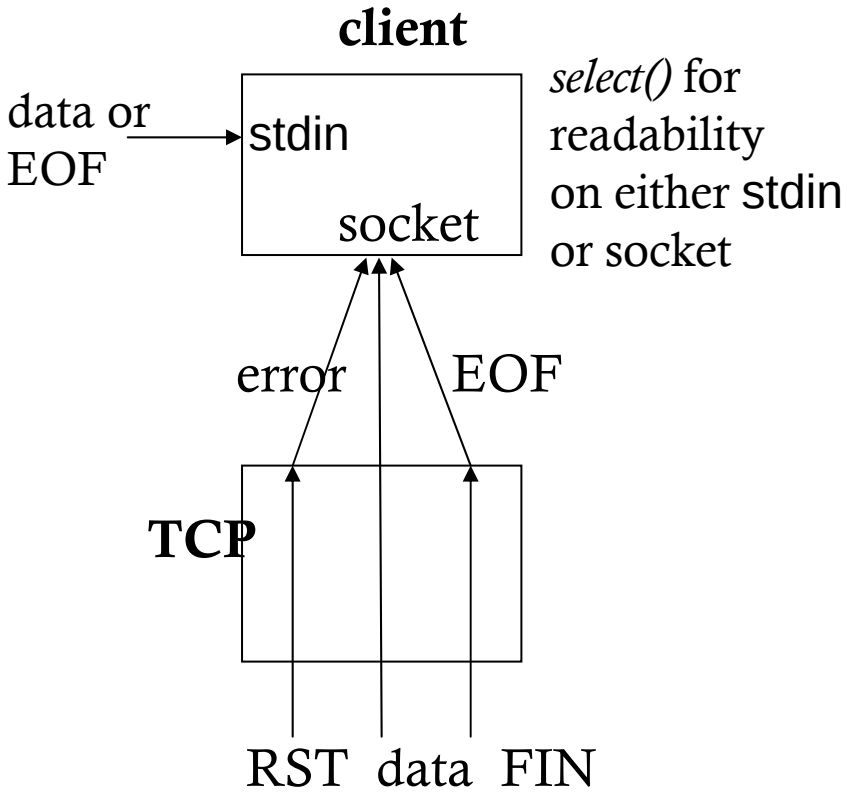
```
 1  #include  "unp.h"

 2  void
 3  str_cli(FILE *fp, int sockfd)
 4  {
 5    int      maxfdp1;
 6    fd_set    rset;
 7    char    sendline[MAXLINE], recvline[MAXLINE];

 8    FD_ZERO(&rset);
 9    for ( ; ; ) {
10      FD_SET(fileno(fp), &rset);
11      FD_SET(sockfd, &rset);
12      maxfdp1 = max(fileno(fp), sockfd) + 1;
13      Select(maxfdp1, &rset, NULL, NULL, NULL);
```



str_cli() Using select() (2/2)

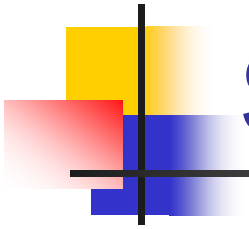
```
14      if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
15          if (Readline(sockfd, recvline, MAXLINE) == 0)
16              err_quit("str_cli: server terminated prematurely");
17          Fputs(recvline, stdout);
18      }

19      if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
20          if (Fgets(sendline, MAXLINE, fp) == NULL)
21              return;    /* all done */
22          Writen(sockfd, sendline, strlen(sendline));
23      }
24    }
25  }
```

Conditions Handled in str_cli()



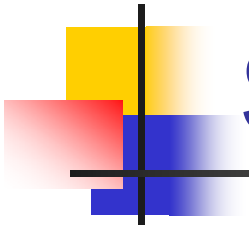
- If the peer TCP sends data, the socket becomes readable, and read() returns greater than 0 (# of bytes read).
- If the peer TCP sends a FIN, the socket becomes readable and read returns 0 (EOF).
- If the peer TCP sends an RST, the socket becomes readable, read() returns -1, and error contains the specific error code.

Ref: UNP, Stevens et. al., vol 1, ed 3, 2004, AW. P. 167.



Stop-and-Wait Mode Operations

- Up to this moment, all versions of str_cli() functions operate in a stop-and-wait mode.
  - A client sends a line to the server and wait for the reply.
  - Time needed for one single request/reply is one RTT plus server's processing time (close to zero for our simple echo server model).
    - One may use the system *ping* program to measure RTTs.
  - It is fine for interactive use, but not a good use of available network bandwidth.
- Use batch-mode operations to better utilize the available high-speed network connections.



Some Assumptions

- Assumptions for illustration purposes:
  - RTT = 8 units of time
  - No server process time (0)
  - Size of request = size of reply
  - Full duplex data transfers

Illustration of Stop-and-Wait

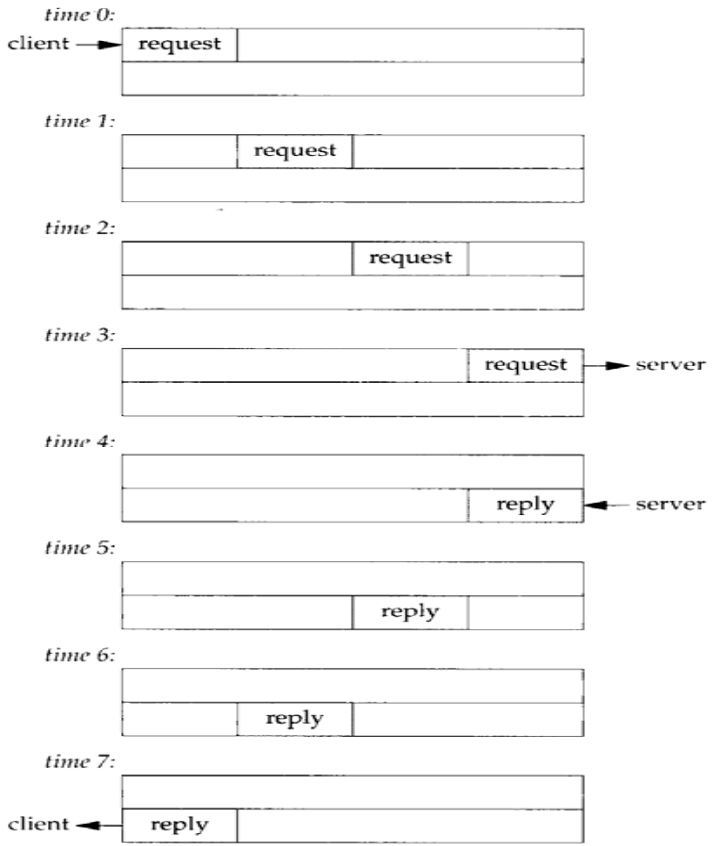


Figure 6.10 Time line of stop-and-wait mode: interactive input.

Illustration of Batch Mode

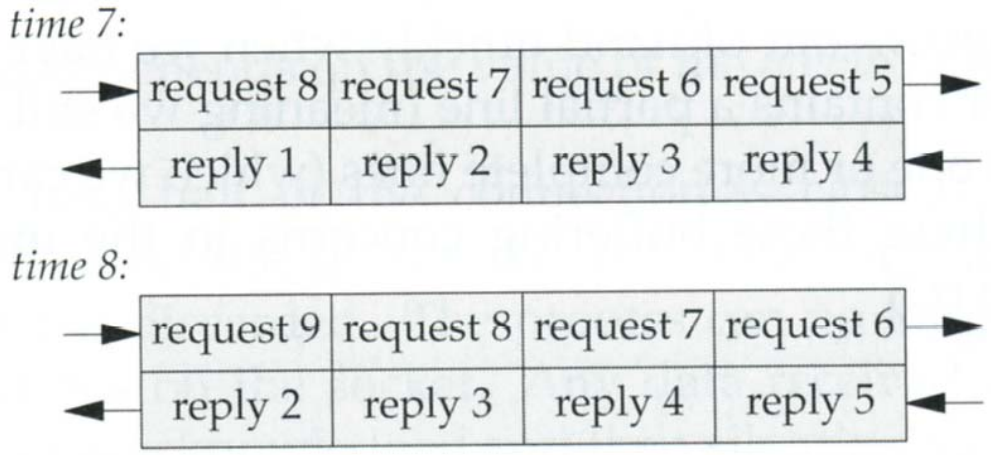
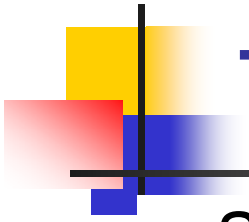


Figure 6.11 Filling the pipe between the client and server: batch mode.

- Note: Batch mode operations can be achieved easily under Unix by just redirecting the standard input and output.



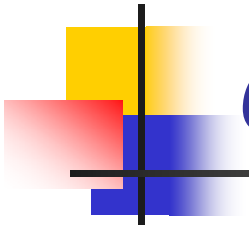
The *shutdown()* Function

- Syntax:

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

int  shutdown(int sockfd, int howto);

                                    Returns: 0 if OK, -1 on error
```

where *howto* has 3 values:

- SHUT_RD
  - The read-half is closed.
- SHUT_WR
  - The write-half is closed.
- SHUT_RDWR
  - Both read and write halves are closed.



close() vs. *shutdown()*

- As mentioned before, close() decrements the descriptor's reference count, and closes the socket, thus terminating both read/write directions of data transfer, if the count reaches 0.
- With shutdown(), we can initiate TCP's normal connection termination sequence regardless of the reference count.

Closing Half a TCP Connection

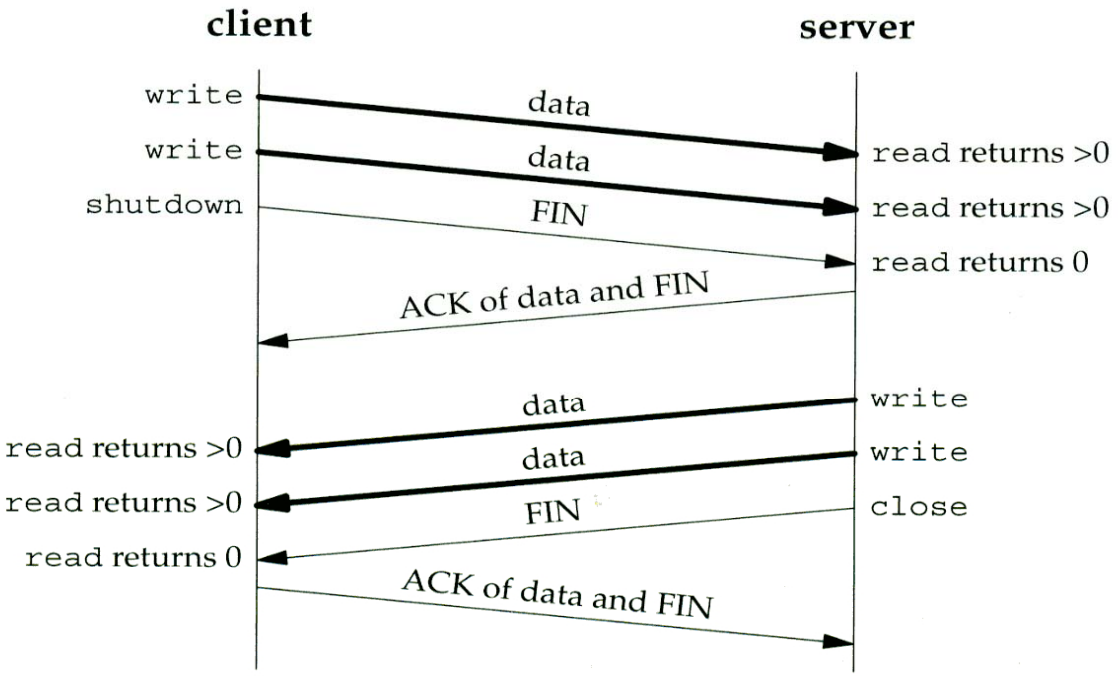


Figure 6.12 Calling `shutdown` to close half of a TCP connection.

str_cli() Revisited (1/2)

- *select/strcliselect02.c*

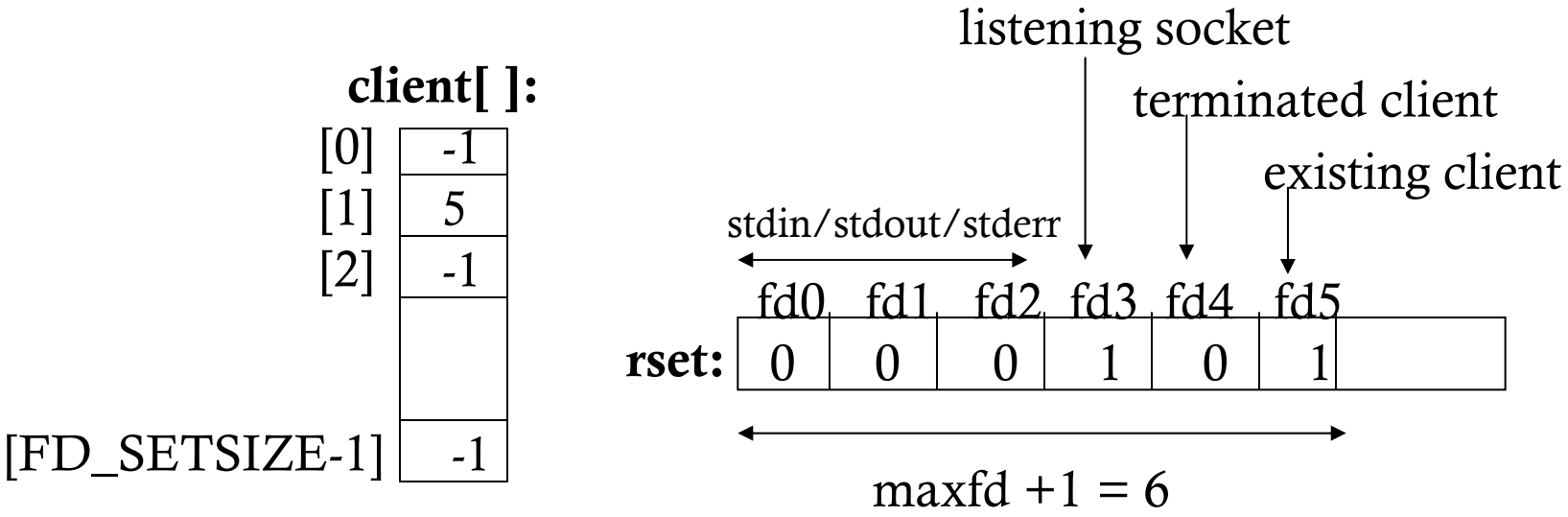
```
 1  #include  "unp.h"
 2  void
 3  str_cli(FILE *fp, int sockfd)
 4  {
 5    int      maxfdp1, stdineof;
 6    fd_set    rset;
 7    char    buf[MAXLINE];
 8    int    n;

 9    stdineof = 0;
10     FD_ZERO(&rset);
11     for ( ; ; ) {
12         if (stdineof == 0)
13             FD_SET(fileno(fp), &rset);
14         FD_SET(sockfd, &rset);
15         maxfdp1 = max(fileno(fp), sockfd) + 1;
16         Select(maxfdp1, &rset, NULL, NULL, NULL);
```

str_cli() Revisited (2/2)

```
17      if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
18          if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
19              if (stdineof == 1)
20                  return;    /* normal termination */
21              else
22                  err_quit("str_cli: server terminated prematurely");
23          }

24          Write(fileno(stdout), buf, n);
25      }

26      if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
27          if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
28              stdineof = 1;
29              Shutdown(sockfd, SHUT_WR);  /* send FIN */
30              FD_CLR(fileno(fp), &rset);
31              continue;
32          }

33          Writen(sockfd, buf, n);
34      }
35  }
36 }
```

TCP Echo Server Revisited (1/5)

- A single server process to handle multiple clients concurrently (using select()).
- In need of some data structures to keep track of the clients.
  - client[] (client descriptor array) and rset (read descriptor set)



TCP Echo Server Revisited (2/5)

- *tcpcliserv/tcpservselect01.c*

```
 1  #include  "unp.h"
 2  int
 3  main(int argc, char **argv)
 4  {
 5    int          i, maxi, maxfd, listenfd, connfd, sockfd;
 6    int          nready, client[FD_SETSIZE];
 7    ssize_t        n;
 8    fd_set       rset, allset;
 9    char         buf[MAXLINE];
10    socklen_t      clilen;
11    struct sockaddr_in  cliaddr, servaddr;
12    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
13    bzero(&servaddr, sizeof(servaddr));
14    servaddr.sin_family      = AF_INET;
15    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16    servaddr.sin_port        = htons(SERV_PORT);
```

TCP Echo Server Revisited (3/5)

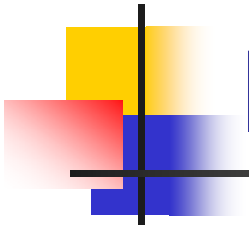
```
17    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

18    Listen(listenfd, LISTENQ);

19    maxfd = listenfd;      /* initialize */
20    maxi = -1;          /* index into client[] array */
21    for (i = 0; i < FD_SETSIZE; i++)
22        client[i] = -1;      /* -1 indicates available entry */
23    FD_ZERO(&allset);
24    FD_SET(listenfd, &allset);

25    for ( ; ; ) {
26        rset = allset;    /* structure assignment */
27        nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

28        if (FD_ISSET(listenfd, &rset)) {  /* new client connection */
29            clilen = sizeof(cliaddr);
30            connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
```

TCP Echo Server Revisited (4/5)

```
31          for (i = 0; i < FD_SETSIZE; i++)
32              if (client[i] < 0) {
33                  client[i] = connfd;  /* save descriptor */
34                  break;
35              }
36          if (i == FD_SETSIZE)
37              err_quit("too many clients");

38          FD_SET(connfd, &allset);  /* add new descriptor to set */
39          if (connfd > maxfd)
40              maxfd = connfd;      /* for select */
41          if (i > maxi)
42              maxi = i;        /* max index in client[] array */

43          if (--nready <= 0)
44              continue;        /* no more readable descriptors */
45      }
```

TCP Echo Server Revisited (5/5)

```
46      for (i = 0; i <= maxi; i++) {  /* check all clients for data */
47          if ( (sockfd = client[i]) < 0)
48              continue;
49          if (FD_ISSET(sockfd, &rset)) {
50              if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
51                  /* connection closed by client */
52                  Close(sockfd);
53                  FD_CLR(sockfd, &allset);
54                  client[i] = -1;
55              } else
56                  Writen(sockfd, buf, n);

57          if (--nready <= 0)
58              break;        /* no more readable descriptors */
59        }
60      }
61    }
62  }
```



DOS Attacks

- Weakness of the TCP echo server — vulnerable to denial-of-service (DOS) attacks.
- Attack scenario:
  - A malicious client sends 1 byte of data (without a newline).
  - Server hangs until the client either sends a newline or terminates.
- Possible solutions:
  - Use nonblocking I/O for the server listening socket.
  - Have each client served by a separate process/thread.
  - Place a timeout on the I/O operations

The *pselect()* Function (1/2)

- The POSIX version of select()

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

int pselect(int maxfdp1,  fd_set  *readset,  fd_set  *writeset,
            fd_set  *exceptset,  const struct timespec  *timeout
            const sigset_t  *sigmask);

                Returns: count of ready descriptors, 0 on timeout, -1 on error
```

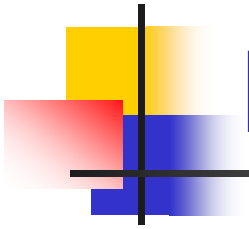
- *timeout*: It records time in seconds and nanoseconds.
- *sigmask*: A pointer to a signal mask, allowing the program some signal handling capabilities.

The *pselect()* Function (2/2)

```
if (intr_flag)
   handle_intr( ); /* handle signal */

if ((nready = select ( ... ))< 0)  {
    if (errno == EINTR) {
        if (intr_flag)
            handle_intr( );
         }
         ....
}
```

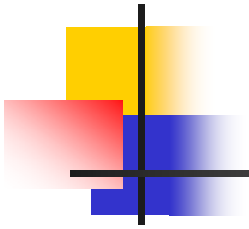
Signal gets lost if select() blocks forever.

```
sigset_t newmask, oldmask, zeromask;

sigemptyset (&zeromask);
sigemptyset (&newmask);
sigaddset (&newmask, SIGINT);

sigprocmask (SIG_BLOCK, &newmask,
     &oldmask);   /*block SIGINT */
if (intr_flag)
    handle_intr( );  /* handle the signal */
if ( (nready = pselect ( ... , &zeromask)) < 0)
     {
      if (errno == EINTR) {
          if (intr_flag)
              handle_intr( );
           }
   ....}
```



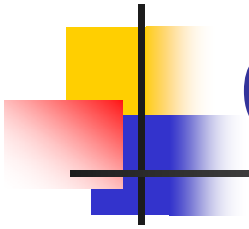
More on Servers

- A server may be designed/developed in several different ways:
  - Iterative
  - Concurrent
  - Preforked
  - Threaded
  - Prethreaded



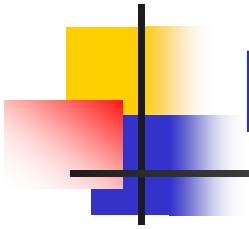
Iterative Servers

- Loop around to serve only one client at a time.
  - Other clients block while one is being serviced.
- It is simple to develop one, but limited usefulness.
  - Only useful for very simple services where the time to serve a client request is very short (e.g., a daytime server).



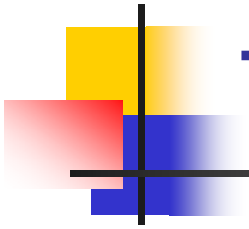
Concurrent Servers

- Use fork() to serve each client.
  - The server will not wait for a client to finish before it starts serving another client.
  - There will be some operating system overhead for fork().
- Good for “medium load” servers.
  - A lot of servers are programmed this way.



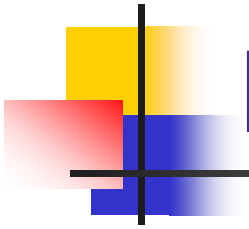
Preforked Servers

- On startup, a server fork()s a configured number of worker processes.
  - When a client connection request arrives, it will be served by an already fork()ed process right away.
- Good for a “heavy load” server.
- The apache Web server is general configured this way.



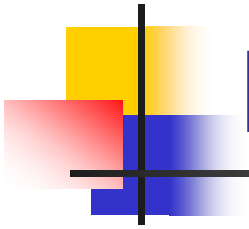
Threaded Servers

- Another type of concurrent servers.
  - Create a thread, instead of using fork(), to handle a client connection request.
- Threads have much lower overhead than processes.
  - Threads are also called light weight processes (LWP).
- Threaded servers may not be portable.
  - Not all systems support threads.



Prethreaded Servers

- Similar concept as preforked servers.
  - Precreate a configured number of worker threads, instead of processes, upon startup.
    - Prethreading amortize the overhead of thread creations all at once, similar to preforking.
- Threaded servers have lower overhead in creation time, but are more complex to deal with than forked servers.
  - Thread synchronization is a major concern in design.



Reading Assignment

- Read Chapters 6, and 16.