Echo client

```c
int sockfd;
struct sockaddr_in server;

socket = Socket(AF_INET, SCOK_STREAM, 0);
server.sin_family = AF_INET;
server.sin_port = htons(SERV_PORT);
Inet_pton(AF_INET, argv[1], &server.sin_addr);
Connect(sockfd, (sockaddr *)&server, sizeof(servaddr));
str_cli(stdin, sockfd);
exit(0);
```

Echo client (cont.)

```c
str_cli(FILE *fp, int sockfd) {
    char sendline[MAXLINE], recvline[MAXLINE];
    while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Writen(sockfd, sendline, strlen(sendline));
        if (Readline(sockfd, recvline, MAXLINE)) == 0
            err_quit("str_cli: server terminated prematurely");
        Fputs(recvline, stdout);
    }
}
```

TCP Termination

1. Server sends FIN
2. Client TCP responds with ACK
3. After that:
   - Server: FIN_WAIT2
   - Client: CLOSE_WAIT
4. The client process is blocked in fgets when FIN arrives on the socket
5. The client is working with two descriptor, while it should not block on one of them:
   - Socket
   - User input

Problem
I/O Multiplexing

- We often need to be able to monitor multiple descriptors:
  - a generic TCP client (like telnet)
  - A server that handles both TCP and UDP
  - Client that can make multiple concurrent requests (browser?)

Example - generic TCP client

- Input from standard input should be sent to a TCP socket.
- Input from a TCP socket should be sent to standard output.

- How do we know when to check for input from each source?

Generic TCP Client

```
while (! done) {
    if ( (n=read(STDIN_FILENO,...)<0))
        if (errno != EWOULDBLOCK)
            /* ERROR */
        else write(tcpsock,...)
    if ( (n=read(tcpsock,...)<0))
        if (errno != EWOULDBLOCK)
            /* ERROR */
        else write(STDOUT_FILENO,...)
}
```

Options

- Use nonblocking I/O.
  - use fcntl() to set O_NONBLOCK
- Use alarm and signal handler to interrupt slow system calls.
- Use multiple processes/threads.
- Use functions that support checking of multiple input sources at the same time.

Non blocking I/O

- use fcntl() to set O_NONBLOCK:

```c
int flags;
flags = fcntl(sock,F_GETFL,0);
fcntl(sock,F_SETFL,flags | O_NONBLOCK);
```

- Now calls to read() (and other system calls) will return an error and set errno to EWOULDBLOCK.
The problem with nonblocking I/O

- Using blocking I/O allows the Operating System to put your process to sleep when nothing is happening (no input). Once input arrives, the OS will wake up your process and read() (or whatever) will return.
- With nonblocking I/O, the process will chew up all available processor time!!!
**struct timeval**

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

```c
struct timeval max = {1,0};
struct timeval forever = NULL;
struct timeval polling = {0,0}
```

**fd_set**

- Implementation is not important
- Operations you can use with an fd_set:

```c
void FD_ZERO( fd_set *fdset);
void FD_SET( int fd, fd_set *fdset);
void FD_CLR( int fd, fd_set *fdset);
int FD_ISSET( int fd, fd_set *fdset);
```

**Using select()**

- Create fd_set
- Clear the whole thing with FD_ZERO
- Add each descriptor you want to watch using FD_SET.
- Call select
- when select returns, use FD_ISSET to see if I/O is possible on each descriptor.

**Errors -- errno**

- EBADF
  - An invalid file descriptor was given in one of the sets.
- EINTR
  - A non blocked signal was caught.
- EINVAL
  - n is negative or the value contained within timeout is invalid.
- ENOMEM
  - select was unable to allocate memory for internal tables.

**shutdown()**

```c
int shutdown( int sockfd, int howto);
```

sockfd is the TCP socket

howto:
- **SHUT_RD**: close the read half of the connection
- **SHUT_WR**: close the write half of the connection
- **SHUT_RDWR**: close both the read and write half of the connection.

shutdown() returns -1 on error (otherwise 0).

**shutdown() vs close()**

- Reference counter:
  - close() decrements the descriptor’s reference count and close the socket only if the count reaches 0.
  - shutdown() initiate TCP connection termination sequence regardless of the reference count.

- Directions:
  - close() terminate both directions of data transfer, reading and writing.
  - shutdown() can close one-half of the TCP connection, either reading or writing.
Assignment & Next time

- Reading:
  - UNP 5.12, 6.3-6.9**

- Next Lecture:
  - thread