**Introduction**

- TCP echo client is handling two inputs at the same time: standard input and a TCP socket
  - when the client was blocked in a call to read, the server process was killed
  - server TCP sends FIN to the client TCP, but the client never sees FIN since the client is blocked reading from standard input
  - We need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.
  - I/O multiplexing (select, poll, or newer pselect functions)

---

**Outline**

- I/O Multiplexing (Chapter 6)
  - Introduction
  - I/O Models
  - Synchronous I/O versus Asynchronous I/O
  - select function
  - TCP echo client using select
  - Shutdown function
  - TCP Echo Server
  - TCP and UDP Echo Server using select (section 8.15)
I/O Models

- Models
  - Blocking I/O
  - Nonblocking I/O
  - I/O multiplexing (`select` and `poll`)
  - Signal driven I/O (`SIGIO`)
  - Asynchronous I/O

- Two distinct phases for an input operation
  - Waiting for the data to be ready (for a socket, wait for the data to arrive on the network, then copy into a buffer within the kernel)
  - Copying the data from the kernel to the process (from kernel buffer into application buffer)
Signal driven I/O (SIGIO)

- Establish SIGIO
- Signal handler
- Signal handler
- recvfrom
- Process
- Datagram ready
- Copy complete
- Kernel
- Process
- Continue executing
- Process blocks while data copied into application buffer
- Process
- Continue executing

Asynchronous I/O

- aio_read System call
- No datagram ready
- Copy data from kernel to user
- Signal handler
- Copy complete
- Process
- Continue executing
- Process
- Continue executing

Comparison of the I/O Models

<table>
<thead>
<tr>
<th>Blocking</th>
<th>Nonblocking</th>
<th>I/O multiplexing</th>
<th>Signal-driven I/O</th>
<th>Asynchronous I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>initiate</td>
<td>check</td>
<td>check</td>
<td>check</td>
<td>initiate</td>
</tr>
<tr>
<td>check</td>
<td>check</td>
<td>blocked</td>
<td>check</td>
<td>wait for data</td>
</tr>
<tr>
<td>check</td>
<td>check</td>
<td>ready</td>
<td>notification</td>
<td>copy data from kernel to user</td>
</tr>
<tr>
<td>check</td>
<td>complete</td>
<td>complete</td>
<td>complete</td>
<td>handles both phases</td>
</tr>
</tbody>
</table>

Synchronous I/O, Asynchronous I/O

- **Synchronous I/O**
  - causes the requesting process to be blocked until that I/O operation (recvfrom) completes.
  - (blocking, nonblocking, I/O multiplexing, signal-driven I/O)
- **Asynchronous I/O**
  - does not cause the requesting process to be blocked
select function

- Allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.
- What descriptors we are interested in (readable, writable, or exception condition) and how long to wait?

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

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

# Possibilities for select function

- Wait forever: return only when descriptor(s) is ready (specify `timeout` argument as `NULL`)
- Wait up to a fixed amount of time
- Do not wait at all: return immediately after checking the descriptors. Polling (specify `timeout` argument as pointing to a `timeval` structure where the timer value is 0)
- The wait is normally interrupted if the process catches a signal and returns from the signal handler
  - `select` might return an error of `EINTR`
  - Actual return value from function = `-1`

Descriptor Sets

- Array of integers: each bit in each integer correspond to a descriptor (`fd_set`)
- 4 macros
  - `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 */

select function Descriptor Arguments

- `readset` ➔ descriptors for checking readable
- `writeset` ➔ descriptors for checking writable
- `exceptset` ➔ descriptors for checking exception conditions (2 exception conditions)
  - arrival of out of band data for a socket
  - the presence of control status information to be read from the master side of a pseudo terminal (Ignore)
- If you pass the 3 arguments as `NULL`, you have a high precision timer than the sleep function
Example of Descriptor sets Macros

```plaintext
fd_set rset;

FD_ZERO(&rset); /*all bits off : initiate*/
FD_SET(1, &rset); /*turn on bit fd 1*/
FD_SET(4, &rset); /*turn on bit fd 4*/
FD_SET(5, &rset); /*turn on bit fd 5*/
```

**maxfdp1 argument to select function**

- specifies the number of descriptors to be tested.
- Its value is the maximum descriptor to be tested, plus one. (hence maxfdp1)
  - Descriptors 0, 1, 2, up through and including maxfdp1-1 are tested
  - example: interested in fds 1,2, and 5 \( \rightarrow \) maxfdp1 = 6
- Your code has to calculate the maxfdp1 value
- constant `FD_SETSIZE` defined by including `<sys/select.h>`
  - is the number of descriptors in the `fd_set` datatype. (often = 1024)

**Value-Result arguments in select function**

- Select modifies descriptor sets pointed to by readset, writeset, and exceptset pointers
- On function call  
  ➢ Specify value of descriptors that we are interested in
- On function return  
  ➢ Result indicates which descriptors are ready
- Use `FD_ISSET` macro on return to test a specific descriptor in an `fd_set` structure  
  ➢ Any descriptor not ready will have its bit cleared
  ➢ You need to turn on all the bits in which you are interested on all the descriptor sets each time you call `select`

**Condition for a socket to be ready for select**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Readable?</th>
<th>writable?</th>
<th>Exception?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data to read</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read-half of the connection closed</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>new connection ready for listening socket</td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>Space available for writing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write-half of the connection closed</td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>Pending error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP out-of-hand data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
str_cli Function revisited

- Recall section 5.5 (source code is $lib/str_cli.c$)
- Problems with earlier version
  - could be blocked in the call to `fgets` when something happened on the socket
  - We need to be notified as soon as the server process terminates
- Alternatively
  - block in a call to `select` instead, waiting for either standard input or the socket to be readable.

Conditions handled with the socket

- Peer TCP sends data
  - the socket becomes readable and `read` returns greater than 0 (number of bytes of data)
- Peer TCP sends a FIN (peer process terminates)
  - the socket become readable and `read` returns 0 (EOF)
- Peer TCP sends a RST (peer host has crashed and rebooted)
  - the socket become readable and returns -1
  - `errno` contains the specific error code
- Source code in `select/strcliselect01.c` tested by `select/tcpcli01.c`
- This version is OK for stop-an-wait mode (interactive input), will modify later for batch input and buffering

select-based str_cli function 1/2

```c
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);
        //Continue.....
    }
}
```
**select-based str_cli function**

if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
if (Readline(sockfd, recvline, MAXLINE) == 0)
err_quit("str_cli: server terminated prem");
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));
}

**Batch Input and Buffering**

- With batch input, can send as fast as we can
- The problem with the revised str_cli function
  - After the handling of an end-of-file on input, the send function returns to the main function, that is, the program is terminated.
  - However, in *batch mode*, there are still other requests and replies in the pipe.
- We need a way to close *one-half of the TCP connection*
  - send a FIN to the server, telling it we have finished sending data, but leave the socket descriptor open for reading
  - shutdown function

**Shutdown function**

- Close one half of the TCP connection
  - send FIN to server, but leave the socket descriptor open for reading
- Limitations with close function
  - decrements the descriptor’s reference count and closes the socket only if the count reaches 0
  - With shutdown, can initiate TCP normal connection termination regardless of the reference count
  - terminates both directions (reading and writing)
- With shutdown, we can tell other end that we are done sending, although that end might have more data to send us
I/O Multiplexing

Shutdown function

<table>
<thead>
<tr>
<th>client</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>data</td>
</tr>
<tr>
<td>write</td>
<td>data</td>
</tr>
<tr>
<td>shutdown</td>
<td>FIN</td>
</tr>
<tr>
<td>ACK of data and FIN</td>
<td>Read returns &gt; 0</td>
</tr>
<tr>
<td>data</td>
<td>Write returns &gt; 0</td>
</tr>
<tr>
<td>FIN</td>
<td>Read returns 0</td>
</tr>
<tr>
<td>write</td>
<td>data</td>
</tr>
<tr>
<td>write</td>
<td>FIN</td>
</tr>
<tr>
<td>close</td>
<td>ACK of data and FIN</td>
</tr>
</tbody>
</table>

#include<sys/socket.h>

int shutdown ( int sockfd, int howto );
/* return : 0 if OK, -1 on error */

• howto argument
  - SHUT_RD
    - read-half of the connection closed
    - Any data in receive buffer is discarded
    - Any data received after this call is ACKed and then discarded
  - SHUT_WR
    - write-half of the connection closed (half-close)
    - Data in socket send buffer sent, followed by connection termination
  - SHUT_RDWR
    - both closed

str_cli using select and shutdown

//Source code in select/strselect02.c, test with select/tcpcli02.c
#include "unp.h"

#define MAXLINE 1024

void str_cli(FILE *fp, int sockfd)
{
    int maxfdp1, stdineof;
    FD_ZERO(&rset);
    char sendline[MAXLINE], recvline[MAXLINE];

    stdineof = 0;
    FD_ZERO(&rset);
    for ( ; ; ) {
        if (stdineof == 0) // select on standard input for readability
            FD_SET(fileno(fp), &rset);
        if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0) {
                if (stdineof == 1)
                    return; /* normal termination */
                else
                    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) {
                stdineof = 1;
            }
            Shutdown(sockfd, SHUT_WR);   /* send FIN */
            if (FD_ISSET(sockfd, &rset)) {
                if (Fgets(sendline, MAXLINE, fp) == NULL) {
                    stdineof = 1;
                    continue;
                }
            }
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
TCP echo server using `select`

- Rewrite the server as a single process that uses `select` to handle any number of clients, instead of forking one child per client.
- Before first client has established a connection

  - `Client[0]`
    - FD_SETSIZE -1
  - rset: fd0 fd1 fd2 fd3
    - fd0(stdin), 1(stdout), 2(stderr)
    - Maxfd + 1 = 4
  - fd:3 listening socket fd

- After first client connection is established (assuming connected descriptor returned by `accept` is 4)

  - `Client[0]`
    - 4
  - `Client[1]`
    - -1
  - `Client[2]`
    - -1
  - rset: fd0 fd1 fd2 fd3 fd4
    - fd:0(stdin), 1(stdout), 2(stderr)
    - fd:3 listening socket fd
    - fd:4 first connected socket fd

- After second client connection is established (assuming connected descriptor returned by `accept` is 5)

  - `Client[0]`
    - -1
  - `Client[1]`
    - 5
  - `Client[2]`
    - -1
  - rset: fd0 fd1 fd2 fd3 fd4 fd5
    - fd:0(stdin), 1(stdout), 2(stderr)
    - fd:3 listening socket fd
    - fd:4 first connected socket fd
    - fd:5 second connected socket fd

- First client terminates its connection (fd 4 readable and read returns 0 → client TCP sent a FIN)

  - `Client[0]`
    - -1
  - `Client[1]`
    - 5
  - `Client[2]`
    - -1
  - rset: fd0 fd1 fd2 fd3 fd4 fd5
    - fd:0(stdin), 1(stdout), 2(stderr)
    - fd:3 listening socket fd
    - fd:5 second connected socket fd
TCP echo server using **select**

- As clients arrive, record connected socket descriptor in first available entry in client array (first entry = -1)
- Add connected socket to read descriptor set
- Keep track of
  - Highest index in client array that is currently in use
  - `Maxfd + 1`
- The limit on number of clients to be served
  - `Min(FD_SETSIZE, Max (Number of descriptors allowed for this process by the kernel))`
- Source code in tcpcliserv/tcpservechoselect01.c

TCP and UDP echo server using **select**

- Section 8.15
- Combine concurrent TCP echo server with iterative UDP server into a single server that uses **select** to multiplex a TCP and UDP socket
- Source code in udpciserv/udpservechoselect01.c
- Source code for `sig_child` function (signal handler) is in udpciserv/sigchildpidwait.c
  - Handles termination of a child TCP server
  - See sections 5.8, 5.9, and 5.10