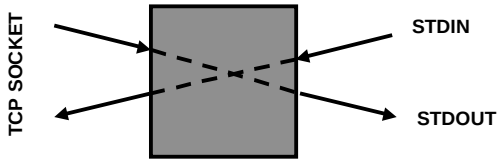
I/O Multiplexing

- We often need to be able to monitor multiple descriptors:
  - a generic TCP client (like telnet)
  - need to be able to handle unexpected situations, perhaps a server that shuts down without warning.
  - A server that handles both TCP and UDP

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



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`**:

```
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`**.


```
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,…)
}
```

The problem with nonblocking I/O

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

Using alarms

```
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(STDIN_FILENO,…);
...
```

A function you write

```
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(tcpsock,…);
...
```


Alarming Problem

What will be the effect on response time ?

What is the ‘right’ value for MAX_TIME?

Select()

- The select() system call allows us to use blocking I/O on a set of descriptors (file, socket, …).
- For example, we can ask select to notify us when data is available for reading on either STDIN or a TCP socket.

select()

```
int select( int maxfd,
            fd_set *readset,
            fd_set *writeset,
            fd_set *excepset,
           const struct timeval *timeout);
```

maxfd: highest number assigned to a descriptor.
weadset: set of descriptors we want to read from.
writeset: set of descriptors we want to write to.
excepset: set of descriptors to watch for exceptions.
timeout: maximum time select should wait

struct timeval

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

struct timeval max = {1,0};
```

`fd_set`

- Implementation is not important
- Operations to use with fd_set:

```
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.