I/O Multiplexing

- Consider an example where a TCP client is handling two inputs at the same time: standard input and a TCP socket.
- It is possible for the client to blocked on a call to read (by calling the ***readline*** function), and the server process will timeout.
- The server TCP correctly sends a FIN to the client TCP, but since the client process is blocked reading from standard input, it never sees the end-of-file until it reads from the socket (possibly much later in time).
- A solution to this is to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output).
- This capability is called **I/O multiplexing** and is provided by the ***select*** and ***poll*** functions.
- I/O multiplexing is typically used in networking applications in the following scenarios:
    - When a client is handling multiple descriptors (normally interactive input and a network socket), 1/0 multiplexing should be used.
    - It is possible, but rare, for a client to handle multiple sockets at the same time. An example of this is a Web client.
    - If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally.
    - If a server handles both TCP and UDP, I/O multiplexing is normally used.
    - If a server handles multiple services and perhaps multiple protocols (e.g., the ***inetd*** daemon), I/O multiplexing is normally used.

Blocking I/O Model

- By default, all sockets are blocking. Consider the UDP example we looked at in our previous discussions.
- A process calls ***recvfrom*** and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
- The process is **blocked** the entire time from when it calls ***recvfrom*** until it returns. When recvfrom returns, the application processes the datagram.

Nonblocking I/O Model

- When we set a socket **nonblocking**, we are telling the kernel that when a requested I/O operation cannot be completed, return an error code instead of blocking.
- Now when we call ***recvfrom***, and there is no data to return, the kernel immediately returns an error of **EWOULDBLOCK** instead of blocking.
- This implies that we have to keep calling ***recvfrom*** until a datagram is received, copied into our application buffer.
- When an application sits in a loop calling ***recvfrom*** on a nonblocking descriptor like this, it is called **polling**.
- The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

- With I/O multiplexing, we call ***select*** or ***poll*** and block in one of these two system calls, instead of blocking in the actual I/O system call.
- We block in a call to ***select***, waiting for the datagram socket to be readable. When ***select*** returns that the socket is readable, we then call ***recvfrom*** to copy the datagram into our application buffer.
- On the surface there does not appear to be any advantage in doing this, and in fact there is a slight disadvantage because using ***select*** requires two system calls instead of one.
- As we shall see, the advantage in using ***select*** is that we can wait for more than one descriptor to be ready. In other words we can select from a **set** of socket descriptors.

Synchronous I/0 versus Asynchronous I/O

- Posix.1 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 I/O models we have discussed, **blocking**, **nonblocking**, and **I/O multiplexing** all synchronous because the actual I/O operation (***recvfrom***) blocks the process.
- Only the true asynchronous I/O model defined by Posix ("**real-time**" extensions) matches the asynchronous I/O definition.

The *select* Function

- This function allows a process to specify to the kernel to notify the process when any one of multiple events occur or when a specified timeout expires.

- If an application is using several sockets then the ***select()*** function may be used to find out which ones are active, i.e. which ones have outstanding incoming data or which can now accept further data for transmission or which have outstanding exceptional conditions.

- This function would probably only be used with server daemon programs using multiplexing to handle multiple clients.

- For example, say we can call select and tell the kernel to return only when:

    - Any of the descriptors in the set {1, 4, 5} are ready for reading or
    - Any of the descriptors in the set {2, 7} are ready for writing or
    - Any of the descriptors in the set {1, 4} have an exception condition pending

- We tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait.

- The descriptors in which we are interested in testing can be any type of descriptor. Select is not restricted to socket descriptors.

- The syntax of the call is as follows:

```
#include &ltsys/time.h>
#include &ltsys/types.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct
                timeval *timeout);
```

- ***select()*** examines the I/O **file descriptor sets** whose addresses are passed in ***readfds***, ***writefds***, and ***exceptfds*** to see if any of their file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively.

- Any of ***readfds***, ***writefds***, and ***exceptfds*** may be given as NULL pointers if no file descriptors are of interest.

- ***nfds*** is the number of bits to be checked in each bit mask that represents a file descriptor; the file descriptors from **0** to ***nfds -1*** in the file descriptor sets are examined.

- On return, ***select()*** replaces the given file descriptor sets with subsets consisting of those file descriptors that are ready for the requested operation.

- The return value from the call to ***select()*** is the number of ready file descriptors.

- If ***timeout*** is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete.

- If timeout is a NULL pointer, the ***select()*** blocks indefinitely. To effect a poll, the ***timeout*** argument should be a non-NULL pointer, pointing to a zero-valued ***timeval*** structure.

- A ***timeval*** structure specifies the number of seconds and microseconds.

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

- There are 3 possibilities:

1. **Wait forever**: return only when one of the specified descriptors is ready for I/O. In this case, the ***timeout*** argument is specified as a null pointer.
2. **Wait up to a fixed amount of time**: return when one of the specified descriptors is ready for I/O, but do not wait beyond the time specified in the ***timeval*** structure.
3. Do not wait at all: Return immediately after checking the descriptors. This is called polling. This is done by setting the timer value to 0.

- A design issue to consider is how to specify one or more descriptor values for each of the three descriptor arguments.

- The ***select*** call uses descriptor sets, typically an **array of integers**, with each bit in each integer corresponding to a descriptor.

- For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63 and so on.

- The implementation details of this variable type are irrelevant as far as the application is concerned and are hidden in the **fd_set** datatype.

- ***select()*** is normally used in conjunction with the macros **FD_SET**, **FD_CLR**, **FD_ISSET** and **FD_ZERO**.

- Before calling ***select***, **FD_SET**, **FD_CLEAR** and **FD_ZERO** should be used to put the relevant descriptors into objects of type **fd_set**.

- For example:

```
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?

 void FD_ZERO(fd_set &fdset);        // clear all bits in fdset
```

- For example, to define a variable of type **fd_set** and then turn on the bits for descriptors 1, 4, and 5, we write:

```
fd_set rset;

FD_ZERO(&rset);    // initialize the set: all bits off
FD_SET(1, &rset);       // turn on the bit for fd 1
FD_SET(4, &rset);       // turn on the bit for fd 4
FD_SET(5, &rset);       // turn on the bit for fd 5
```

- ***select*** modifies the descriptor sets pointed to by the **readset**, **writeset**, and **exceptset** pointers.

- When we call the function, we specify the values of the descriptors that we are interested in and upon return the result indicates which descriptors are ready.

- We use **FD_ISSET** to test a specific descriptor in an **fd_set** structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set.

- To handle this we turn on all the bits in which we are interested in all the descriptor sets each time we call ***select***.

- The return value from this function indicates the total number of bits that are ready across all descriptor sets.

Descriptor Conditions

- From the above discussion we know that a process will wait for a descriptor to become ready for I/O (reading or writing) or to have an exception condition pending on it (out-of-band data).

- I/O for regular files is relatively straightforward. For sockets however we must be more specific about the conditions that cause select to return "ready" for sockets. These conditions are described below.

- A socket is ready for **reading** if any of the following **four conditions** is **true**:

    1. 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.

       We can set this low-water mark using the **SO_RCVLOWAT** socket option. It defaults to I for TCP and UDP sockets.

    2. The read-half of the connection is closed (i.e., a TCP connection that has received a **FIN**). A read operation on the socket will not block and will return 0 (i.e., end-of-file).

    3. The socket is a listening socket and the number of completed connections is nonzero.

    4. 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.

       These **pending errors** can also be fetched and cleared by calling ***getsockopt*** specifying the **SO_ERROR** socket option.

- A socket is ready for **writing** if any of the following **three conditions** is **true**:

5. 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 **and** either (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP).

   This means that if we set the socket **nonblocking**, a write operation will not block and will return a positive value (bytes received).

   We can set this low-water mark using the **SO_SNDLOWAT** socket option. This low-water mark normally defaults to 2048 for TCP and UDP sockets.

6. The write-half of the connection is closed. A write operation on the socket will generate **SIGPIPE**.

7. 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*** for the **SO_EPROR** socket option.

- A socket has an **exception condition pending** if there exists **out-of-band data** for the socket or the socket is still at the **out-of-band mark**.

- Note that when an error occurs on a socket it is marked as both **readable** and **writable** by ***select***.

- The purpose of the receive and send low-water marks is to give the application control over how much data must be available for reading or how much space must be available for writing before ***select*** returns readable or writable.

- For example, if we know that our application has nothing productive to do unless at least 64 bytes of data are present, we can set the receive low-water mark to 64 to prevent select from waking us up if less than 64 bytes are ready for reading.

- As long as the send low-water mark for a UDP socket is less than the send buffer size (which should always be the default relationship), the UDP socket is always writable, since a connection is not required.

TCP Example using *select*

- We will use the basic **echo** server as an example. The server will be a single process that uses **select** to handle multiple clients.

- Let us first examine the data structures that are used to keep track of the clients. The first diagram shown shows the state of the server with just one listening socket open.

- The server maintains a read descriptor set as shown in the next diagram. We assume that the server is started in the foreground, so descriptors **fd0**, **fd1**, and **fd2** are set to standard **input**, **output**, and **error**.

- Therefore the first available descriptor for the listening socket is **fd3**.

- Also shown is an array of integers named **client** that contains the connected socket descriptor for each client. All elements in this array are initialized to -1.

- The only nonzero entry in the descriptor set is the entry for the **listening socket** and the first argument to ***select*** will be ***4***.

- When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls ***accept***.

- The new connected descriptor returned by ***accept*** will be ***fd4***, given the assumptions of this example. The next diagram shows the connection from the client to the server.

- From this point on the server must remember the new connected socket in its ***client*** array, and the connected socket must be **added to the descriptor set**. The updated data structures are shown.

- No assume that some time later a second client establishes a connection and we have the scenario shown. The new connected socket (which we assume is ***fd5***) must be stored, giving the updated data structures shown.

- Next we assume the first client terminates its connection. The client TCP sends a **FIN**, which makes descriptor 4 in the server readable.

- We then close this socket and update our data structures accordingly The value of ***client[0]*** is set to ***-1*** and descriptor ***fd4*** in the descriptor set is set to ***0***.

- This is shown in the diagram. Notice that the value of ***maxfd*** does not change.

- To summarize, as clients arrive we record their connected socket descriptor in the first available entry in the **client** array (i.e., the first entry with a value of -1).

- We must also **add** the connected socket to the **read descriptor set**. The variable maxi is the highest index in the **client** array that is currently in use and the variable ***maxfd*** (plus one) is the current value of the first argument to ***select***.

- The only limit on the number of clients that this server can handle is the minimum of the two values **FD_SETSIZE** and the maximum number of descriptors allowed for this process by the kernel