One way of providing concurrency in the server to handle multiple clients at the same time was calling the *fork()* function. This week we will emphasize the alternative method of handling multiple clients simultaneously: **select** function.

Select **Function:**

This function allows a process to specify a list of descriptors (**ports**), and to instruct the kernel to wait for any of these specified events to occur; and to wake up the process only when one or more of these events occur or when a specified amount of time has passed.

As an example, we can call *select* and tell the kernel to return only when:

- Any of the descriptors in a specified set of sockets are ready for reading
- Any of the descriptors in a specified set of sockets are ready for writing
- Any of the descriptors in a specified set of sockets have an exception condition pending
- *Timeval* (a specified limit of time) seconds have elapsed.

To use *"select"*, we tell the kernel **what descriptors we are interested in** (for reading, writing, or an exception condition) and **how long to wait**. *Select()* **suspends the program until** one of the descriptors in the list becomes ready to perform I/O and **returns an indication of which descriptors are ready**. (establishment of a new client connection, arrival of data, FIN, etc.) Then the program can proceed with I/O on that descriptor, and the operation does not block until I/O is finished.

Here is the function syntax:

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

int select ( int maxFileDescriptorsPlus1, fd_set * readDescriptors, fd_set * writeDescriptors, fd_set *
exceptionDescriptors, struct timeval * timeout );


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

The last argument tells the kernel how long to wait for one of the specified descriptors to become ready. *Timeval* structure specifies the number of seconds and microseconds.

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

There are three possibilities:

i) WAIT FOREVER: Return **only** when one of the specified descriptors is ready for I/O. For this, specify the *timeout* value as a **null pointer.**

ii) WAIT FOR A FIXED AMOUNT OF TIME: Return when one of the specified descriptors is ready for I/O; but **do not wait beyond** the number of seconds and microseconds specified in the *timeval* structure pointed to by the *timeout* argument.

iii) DO NOT WAIT AT ALL: **Return immediately** after checking the descriptors. This is called **polling**. To specify this, the *timeout* argument must point to a *timeval* structure with timer values = 0.

!!! The wait in the first two scenarios is normally interrupted if the process catches a signal and returns from the signal handler.

The descriptor sets that *select* uses are typically 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. All the implementation details are independent of the application and are implicit in the **fd_set** data type and in the following 4 macros:

fd: File descriptors, **fdset**: File descriptor Set

```
void FD_ZERO( fd_set * fdset );          /*clear all bits in the file descriptor set */
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 ? */
```

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

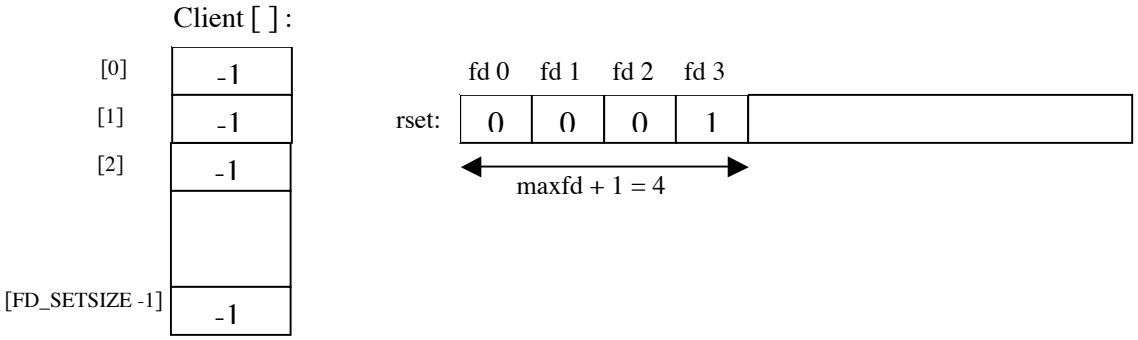
```
fd_set rset;

FD_ZERO (&rset);        /* initialize the set: all bits off —IT IS IMPORTANT TO INITIALIZE THE SET !! */
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*/
```

The argument *maxFileDescriptorsPlus1* specifies the number of descriptors to be tested. Its value is the maximum descriptor to be tested plus one. Hence the descriptors 0, 1, 2,.., (maxFdplus1)-1 are tested.

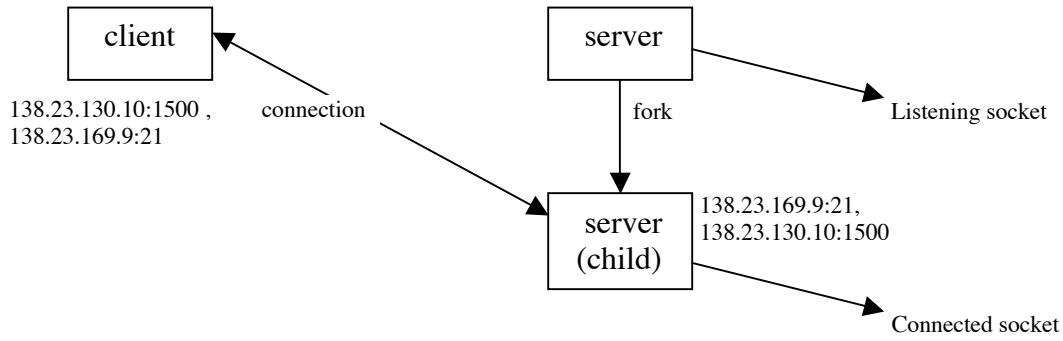
select **modifies** the descriptor sets pointed by the readDescriptors, writeDescriptors, exceptionDescriptors. When we call the function we specify these parameters and on return the function notifies us of which descriptors are ready. We use the FD_ISSET macro on return from select() call in order to test a specific descriptor in an fd_set structure.

Now we can revisit our concurrent TCP server and rewrite this server as a **single** process that uses *select* to handle multiple clients, instead of **fork**ing one child per client.

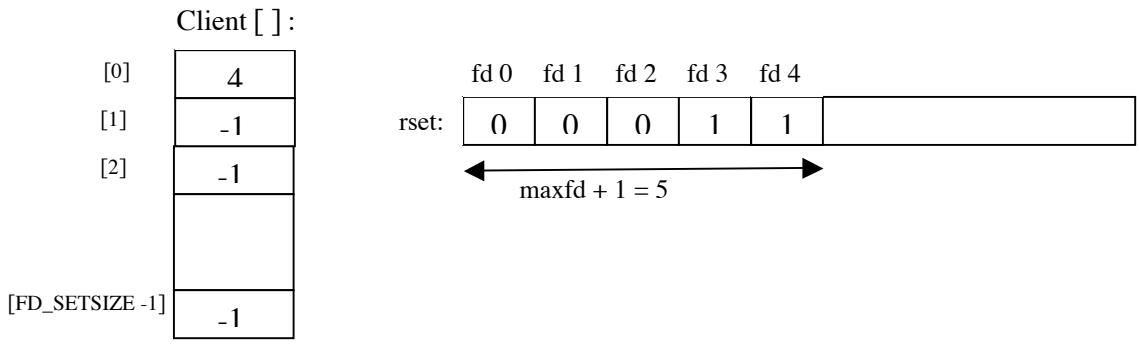
The server has a single listening descriptor.
The server maintains only a read descriptor set as shown below:



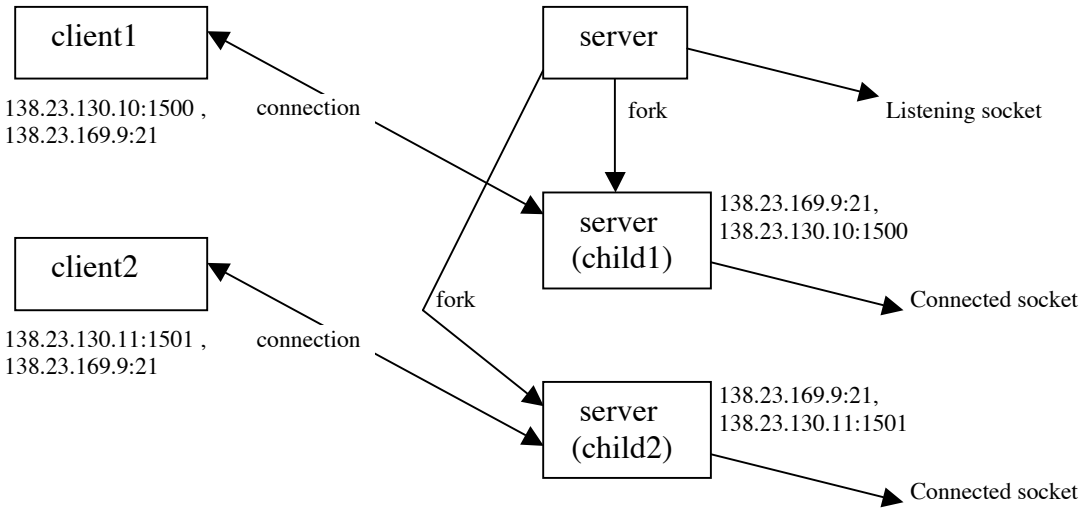
It is assumed that the server is started in the foreground; therefore descriptors 0,1,2 are set to standard input, output, error respectively. So the first available descriptor for the listening socket is 3. "Client []" is an array of integers and it contains the connected socket descriptor for each client. All elements in this array are initialized to –1. The only nonzero entry in the descriptor set is the entry for the listening sockets and the first argument to select will therefore be 4.



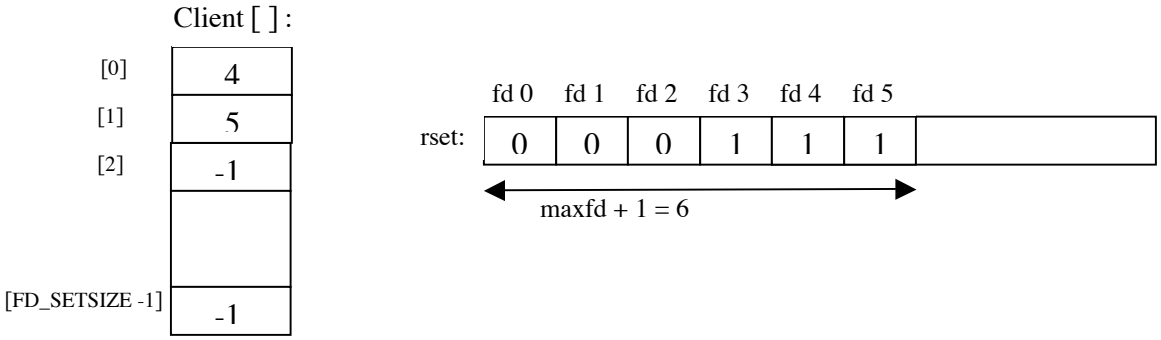
When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls *accept()*. The scenario is shown in the above figure. The new connected descriptor returned by *accept()* will be "4" in this example. From this point on, the server must remember the new connected socket in its *client* array, and the connected socket must be added to the descriptor set. The updated condition is shown below:



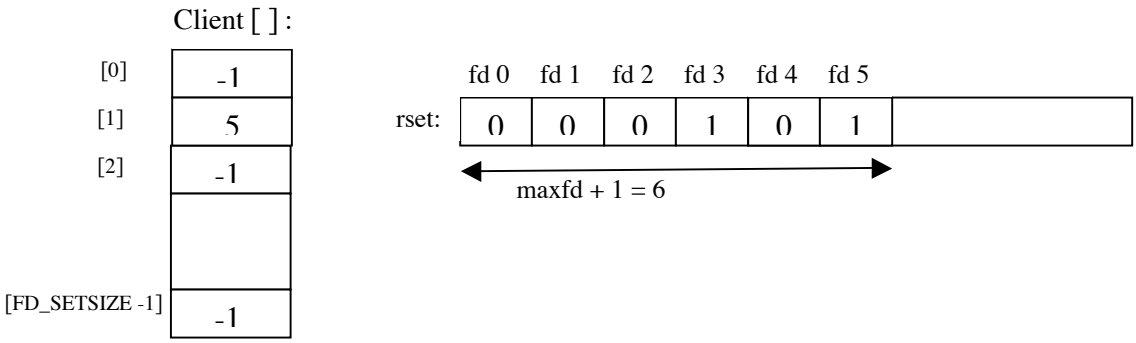
Some time later when a second client establishes a connection and we have the scenario as shown in figure below:



And the data structures after the second client establishes communication will look like this:



When the first client terminates its connection, the server then closes this socket and the data structures are updated accordingly on the server. The value of *client[0]* is set to –1 and descriptor 4 in the dsescriptor set is set to 0. Notice that the value of *maxfd* does not change.



In summary, as clients arrive, we record their connected socket descriptor in the first available entry in the *client* array (i.e. the first entry with value –1). We also add the connected socket to the *read descriptor set*. The variable *maxfd* (plus 1) 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 2 values FD_SETSIZE and the maximum number if descriptors allowed for this process by the kernel.