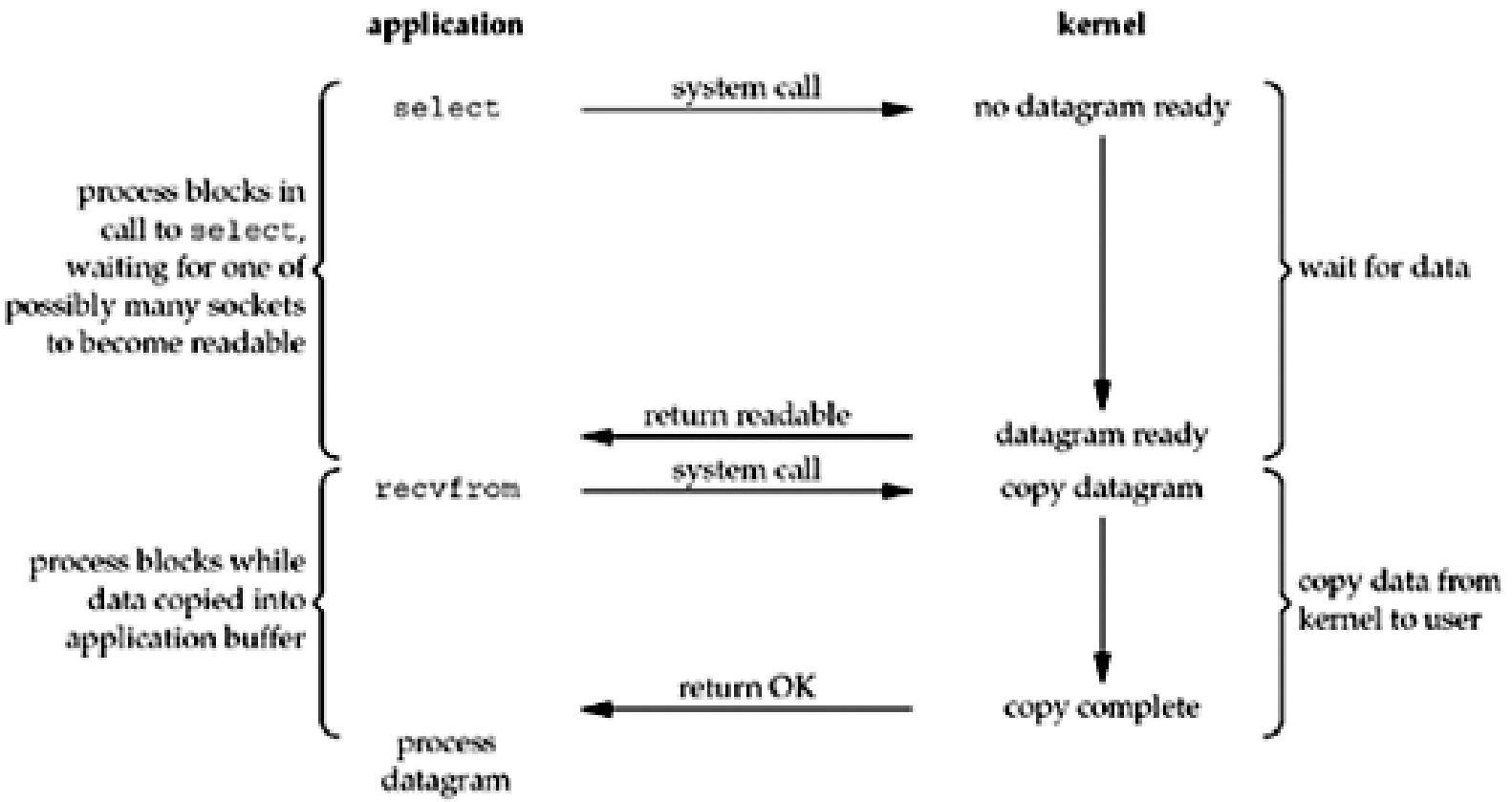
I/O MULTIPLEXING SERVER

Tong Van Van, SoICT, HUST

Content

- I/O Multiplexing Model
- select()
- poll()

I/O Multiplexing Model



select()

- The **select()** function asks kernel to simultaneously check multiple sockets to see if they have data waiting to be **recv()**, or if you can **send()** data to them without blocking, or if some exception has occurred.
- The kernel to wake up the process only when one or more of events occurs or when a specified amount of time has passed.
- Exp : kernel to return only when
  - {1, 4, 5} are ready for reading
  - {2, 7} are ready for writing
  - {1, 4} have an exception condition pending
  - 10.2 seconds have elapsed

select()

```
#include <sys/select.h>
int select(int maxfd, fd_set *readfds, fd_set *writefds,
                fd_set *exceptfds, struct timeval *timeout);
```

- [IN] **`maxfd`** SHOULD BE the highest-numbered file descriptor in any of the three sets, plus 1
- [IN] **`readfds, writefds , exceptfds`** : set of FD to test for reading/receiving, writing/sending, exception conditions. Aleast one of them SHOULD BE not `NULL`.
- [IN] **`timeout`** : how long to wait for one of the specified descriptors to become ready. There are three types of using timeout
  - **`NULL`**: Wait forever
  - Wait up to a fixed amount of time
  - Do not wait at all : **`timeout`** points to the a `timeval` structure having the value 0
- Return:
  - On success, returns the total number of bits that are set(that is the number of ready file descriptors)
  - On time-out, returns 0
  - On error, return -1

fd_set

- Other *fd_sets* use to specify the descriptors that we want the kernel to test for reading, writing, and exception conditions.
- To specify one or more descriptor values for each of these three arguments, select uses *descriptor sets*.
- All the implementation details are irrelevant to the application and are hidden in the *fd_set* datatype and the following four 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); /* Return true if fd is in the fdset */
```

`select()` - Conditions

- Ready for reading:
  - The socket send buffer is not empty
  - The read half of the connection is closed
  - The listening socket receives a new connection request
  - A socket error is pending
- Ready for writing:
  - The size of the available space in the socket send buffer and either: (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP).
  - The write half of the connection is closed
  - A socket using a non-blocking `connect` has completed the connection, or the `connect` has failed
  - A socket error is pending
- Exception: TCP out-of-band data

`select()` - How does kernel work?(Simplifying)

Input: maxfd, readfds, writefds, exceptfds, timeout

```
retval = 0;
is_timeout = false;
while(1){
    check_ready();
    is_timeout = check(timeout);
    if(retval || timeout ||
                interrupt()) break;
       if (check_error()){
          retval = err_num;
          break;
       }
}
wakeup_process();
return retval;
```

```
check_ready(){
  for(i = 0; i < maxfd, i++){
    in = readfds[i-th bit];
    out = writefds[i-th bit];
    ex = exceptfds[i-th bit];

    if(in | out | ex) continue;

    res_in = check_ready_in(i);
    if(res_in){
      readfds[i-th bit] = in & res_in;
      retval++;
    }
    res_out = check_ready_out(i);
    if(res_out){
        writefds[i-th bit] = out & res_out;
        retval++;
      }
    res_ex = check_ready_ex(i);
    if(res_out){
        exceptfds[i-th bit] = out & res_ex;
        retval++;
      }
   }
}
```

Examples

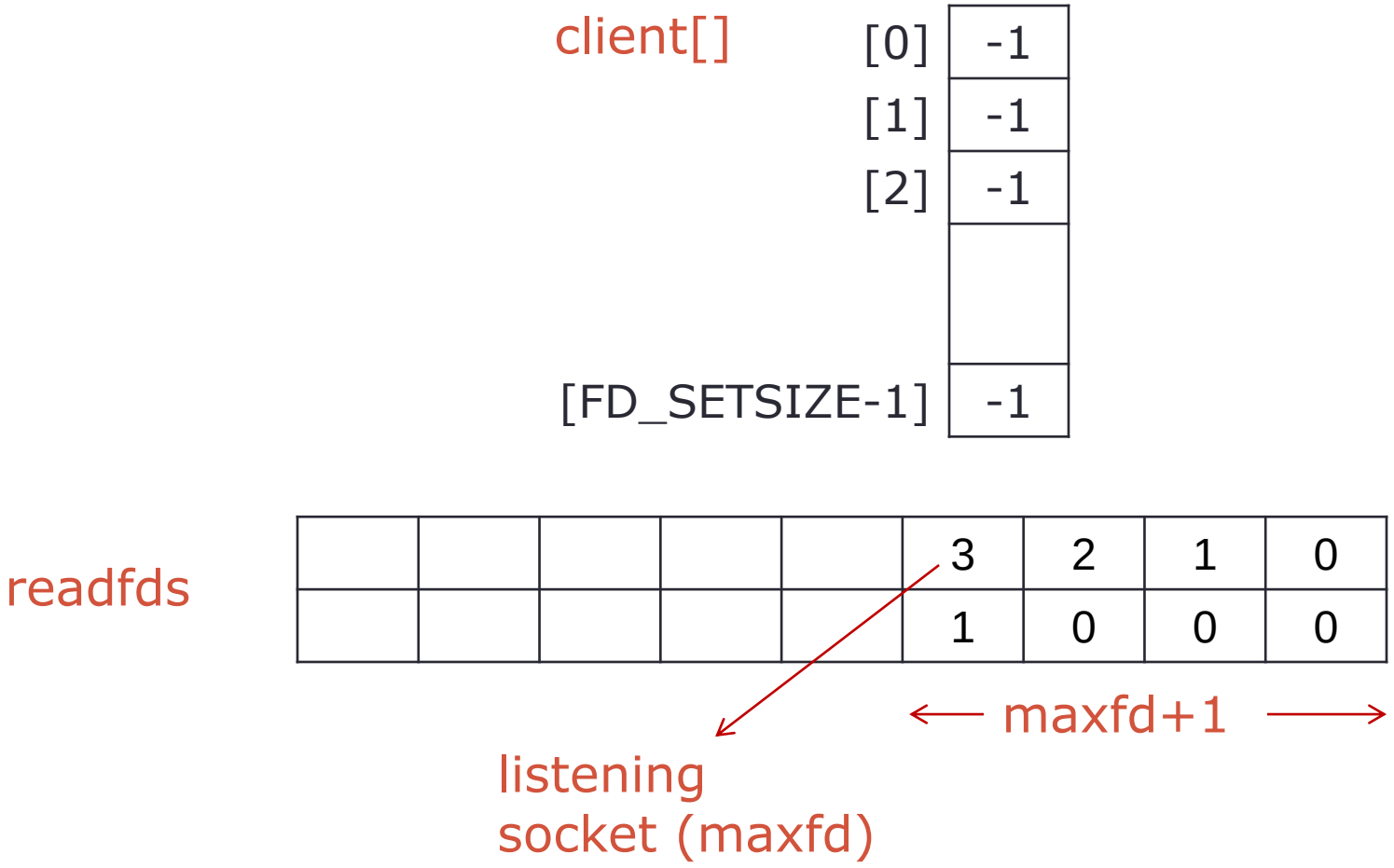
```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];
//pretend we've connected both to a server at this point
//s1 = socket(...); s2 = socket(...);
//connect(s1, ...)...   connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);

// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// since we got s2 second, it's the "greater", so we use
that for
// the n param in select()
n = s2 + 1;
```

Examples (2)

```
// wait until either socket has data ready to be recv()d
//(timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);
if (rv == -1) {
   perror("\Error: "); // error occurred in select() }
else if (rv == 0)
   printf("Timeout occurred! No data after 10.5s \n");
else {
   // one or both of the descriptors have data
   if (FD_ISSET(s1, &readfds))
       recv(s1, buf1, sizeof buf1, 0);
   if (FD_ISSET(s2, &readfds))
       recv(s1, buf2, sizeof buf2, 0);
}
```

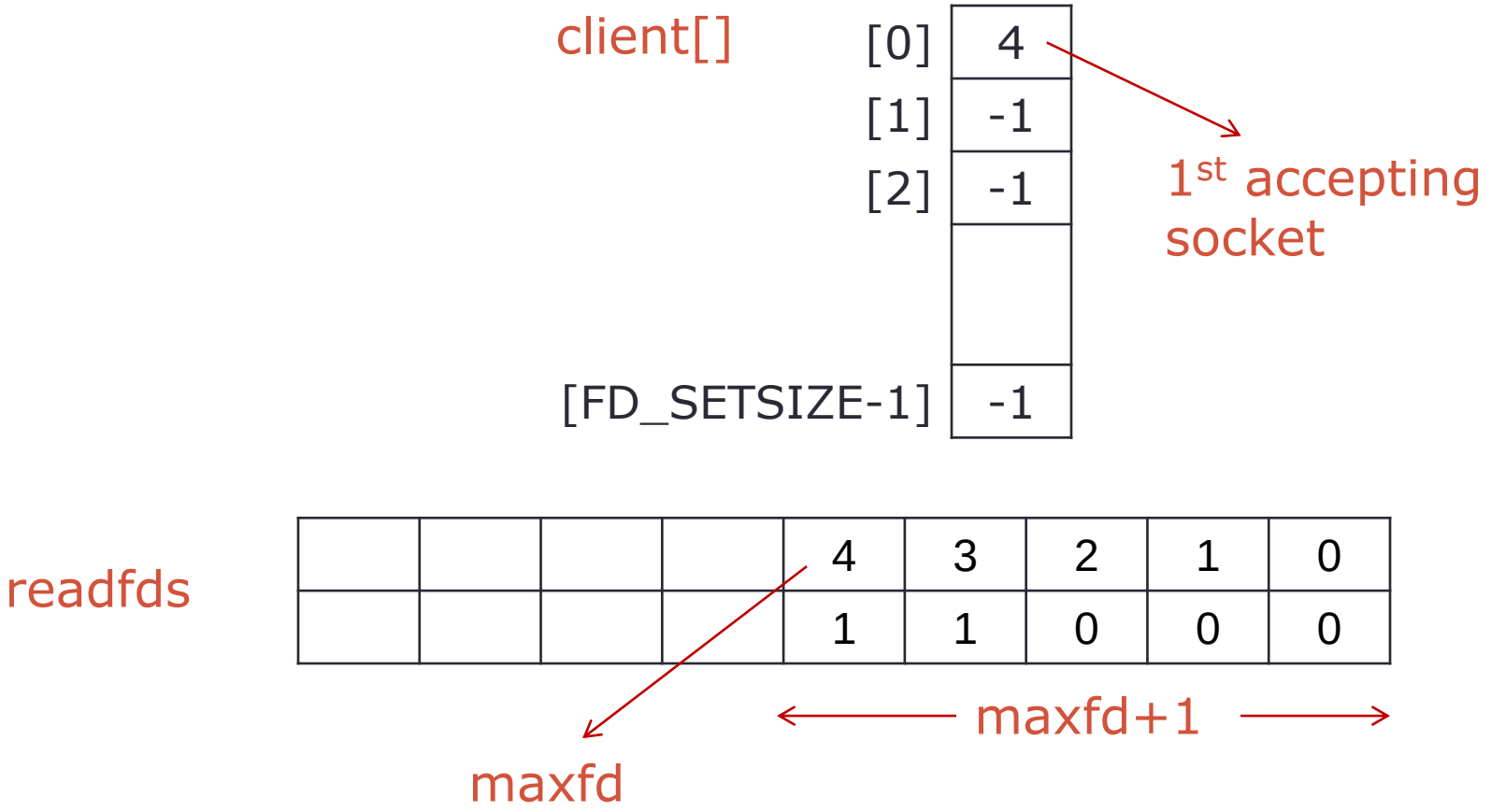
How to use select() in TCP server

Data structures for TCP server with just a listening socket



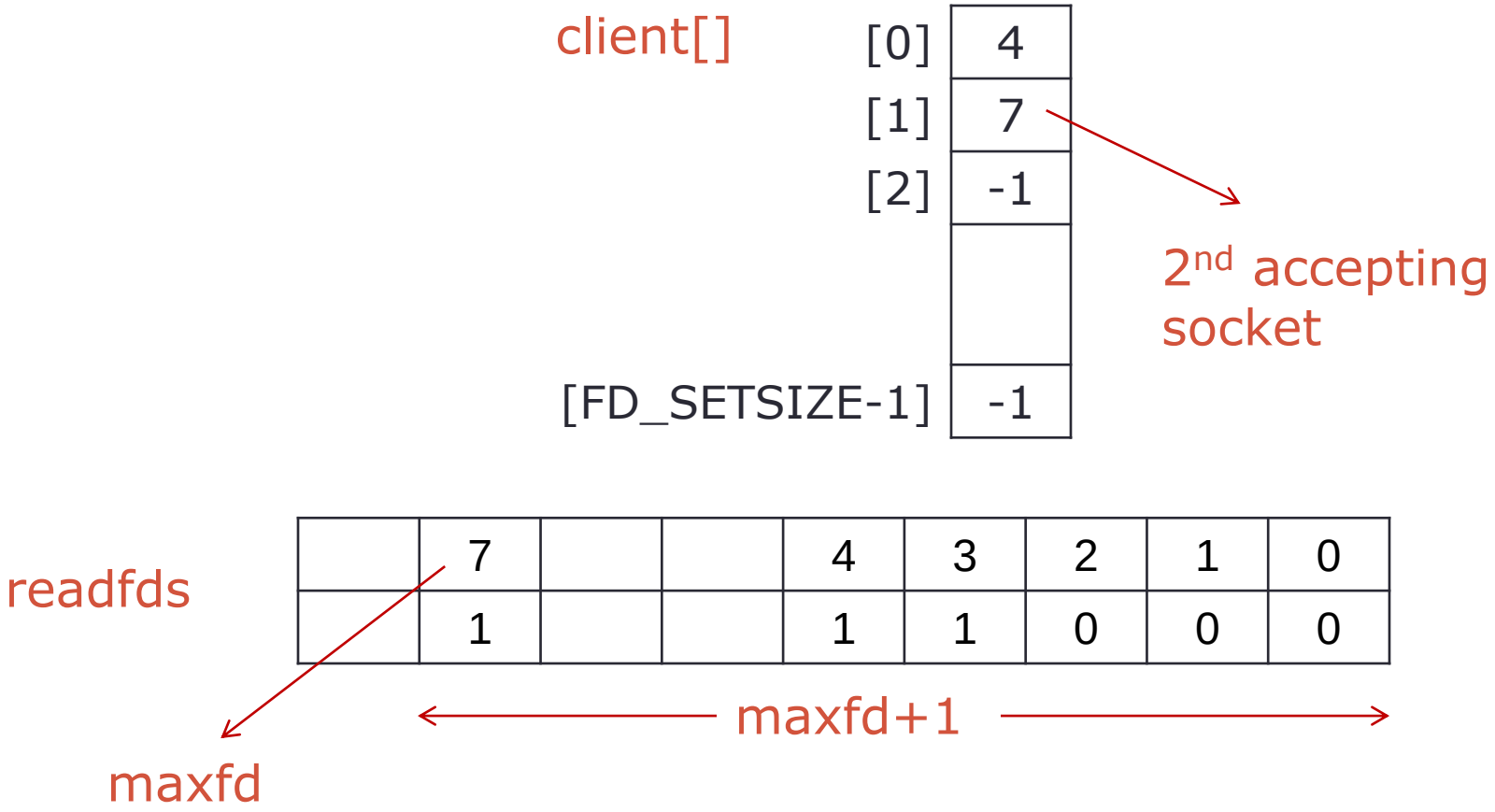
How to use select() in TCP server

Data structures after the 1st client connection is established



How to use select() in TCP server

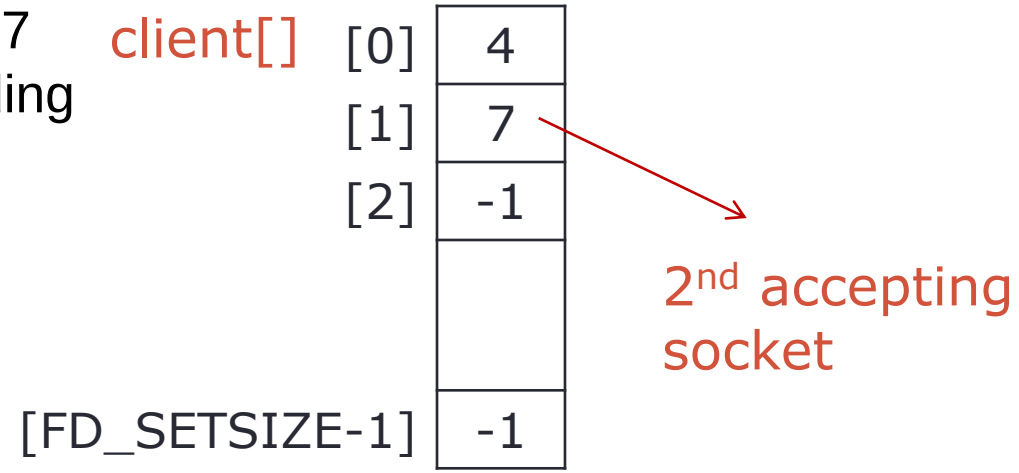
Data structures after the 2nd client connection is established

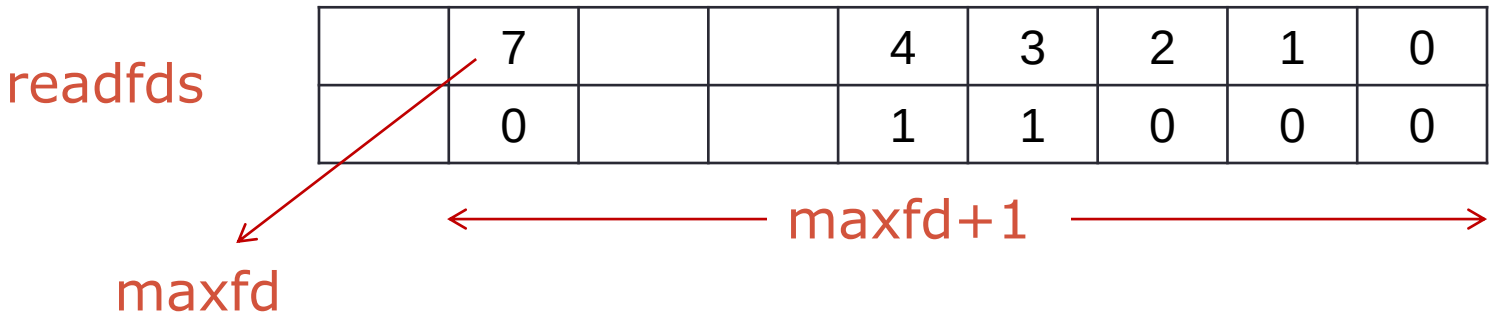


How to use select() in TCP server

After `select()` return. Example:

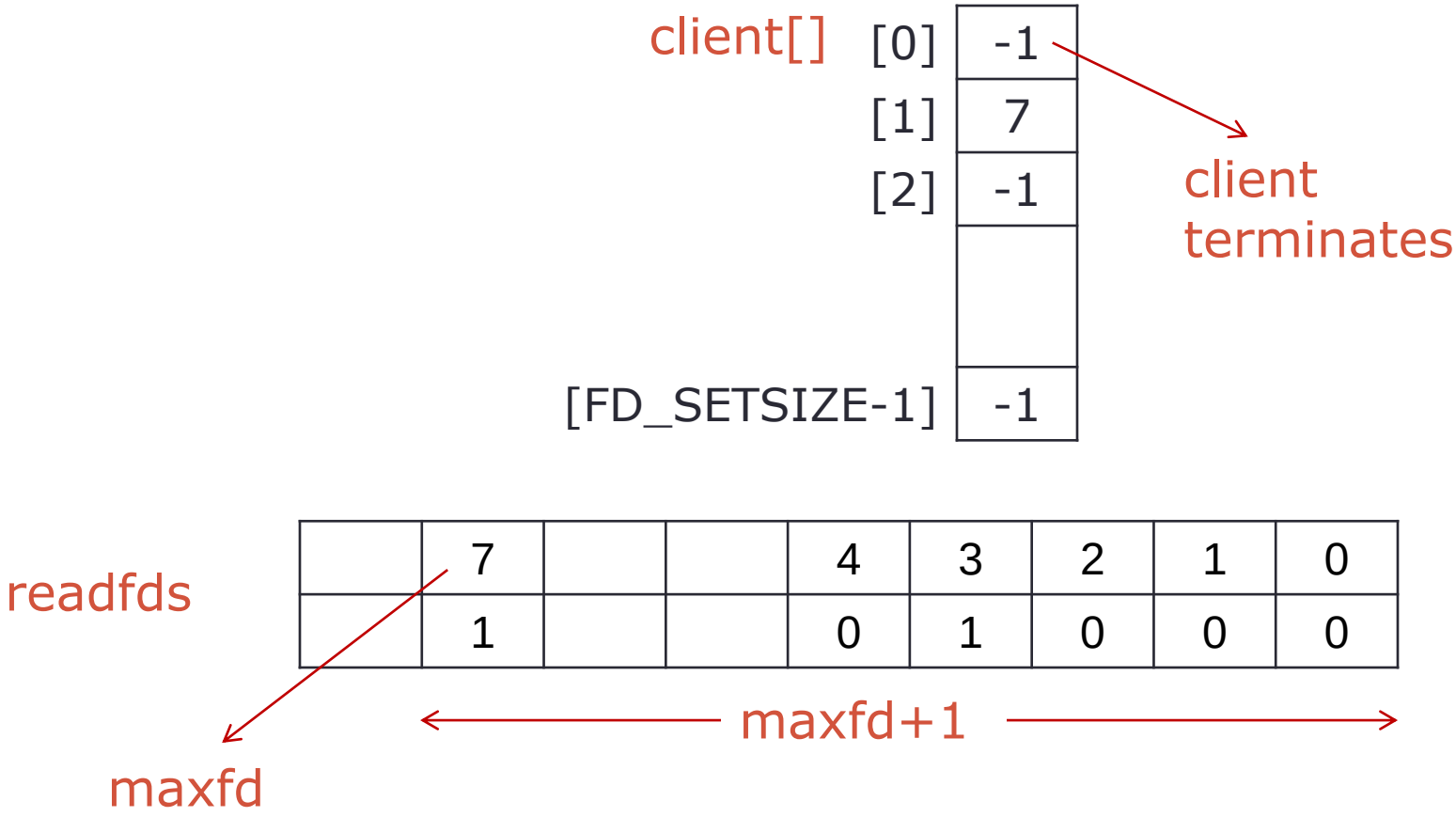
- New connection has established
- No data arrival on socket 7
- Socket 4 is ready for reading





How to use select() in TCP server

Data structures after a client terminates its connection



How to use select() in TCP server

```
//Step 1: Construct socket
//Step 2: Bind address to socket
//Step 3: Listen request from client

//Step 4: Initiate data structures
//Assign initial value for the array of connection socket
for(…) client[i] = -1;

//Assign initial value for the fd_set
//checkset: set of file descriptors that tested by kernel
fd_set checkfds, readfds, writefds, exceptfds;
FD_ZERO (&chkfds);

//Set bit for listenfd
FD_SET(listenfd, &chkfds)
maxfd = listenfd;
```

How to use select() in TCP server

```
//Step 5: Accept connection and communicate with clients
while(…){
    readfds = writefds = exceptfds = checkfds;
    nEvents = select(…);
    //check the status of listenfd
    if(FD_ISSET(listenfd,…)){
        connfd = accept(…);
        FD_SET(connfd, &checkfds);
        if(connfd > maxfd) maxfd = connfd;
        for(…)
            if (client[i] == -1) client[i] = connfd;
    }
    //check the status of connfd(s)
    for(…){
        if(FD_ISSET(client[i],…)){
            doSomething();
            close(connfd);
            client[i] = -1;
            FD_CLEAR(client[i],…)
        }
}
```

TCP Echo Server(revisited - 1)

```
int client[FD_SETSIZE], connfd;
fd_set checkfds, readfds;
sockaddr_in clientAddr;
int ret, nEvents, clientAddrLen;
char rcvBuff[BUFF_SIZE], sendBuff[BUFF_SIZE];

//Step 4: Initiate data structures
for(int i = 0; i < FD_SETSIZE; i++)
   client[i] = -1;

FD_ZERO(&checkfds);

FD_SET(listenfd, &checkfds)
maxfd = listenfd;
```

TCP Echo Server(revisited - 2)

```
//Step 5: Accept connection and communicate with clients
while(1){
   readfds = checkfds;
   nEvents = select(maxfd + 1, &readfds, NULL, NULL, NULL);
   if(nEvents < 0){
       perror("\nError:");
       break;
   }

   if(FD_ISSET(listenfd, &readfds)){       //new client
       clientAddrLen = sizeof(clientAddr);
       connfd = accept(listenfd, (sockaddr *)
                       &clientAddr, &clientAddrLen);
       int i;
       for(i = 0; i < FD_SETSIZE; i++)
          if(client[i] <= 0){
              client[i] = connfd;
              break;
          }
```

TCP Echo Server(revisited - 3)

```
        if(i == FD_SETSIZE)
            printf("\nToo many clients.");
        if(--nEvents <=0)continue; //no more event
    }
    //check the status of connfd(s)
    for(int i = 0; i < FD_SETSIZE; i++){
        if(client[i] <= 0) continue;
        if(FD_ISSET(client[i], &readfds)){
            ret = receiveData(client[i], rcvBuff, BUFF_SIZE, 0);
            if (ret <= 0){
                FD_CLR(client[i], &readfds);
                closesocket(client[i]);
                client[i] = 0;
            }
            else if(ret > 0){
                processData(rcvBuff, sendBuff);
                sendData(client[i], sendBuff, ret, 0);
            }
            if(--nEvents <=0)continue; //no more event
        }
     }
} //end while
```

TCP Echo Server(revisited - 4)

```
/* The processData function copies the input string to output*/
void processData(char *in, char *out){
   strcpy (out, in);
}
/* The recv() wrapper function*/
int receiveData(int s, char *buff, int size, int flags){
   int n;
   n = recv(s, buff, size, flags);
   if(n < 0)
        perror("Error: ");
   return n;
}
/* The send() wrapper function*/
int sendData(int s, char *buff, int size, int flags){
   int n;
   n = send(s, buff, size, flags);
   if(n < 0)
        perror("Error: ");
   return n;
}
```

poll()

```
#include <poll.h>
int poll (struct pollfd *fdarray, unsigned long nfds,
                                  int timeout);
```

- Similar to `select()`
- Provides additional information when dealing with STREAMS devices
- Parameter:
  - `fdarray`: point to the array of `pollfd` structures
  - `nfds`: number of elements in `fdarray`
  - `timeout`: INFTIM(wait forever), 0(return immediately) or >0(wait specified number of milliseconds)
- Return: number of elements have had event, 0 if timeout, -1 if error

pollfd structure

```
struct pollfd {
    int fd;          // the socket descriptor
    short events;    // bitmap of events we're interested in
    short revents;   // when poll() returns, bitmap of events
                     //that occurred
};
```

- events and revents are bitmasks constructed by OR'ing a combination of the following event flags

Constant	events	revents	Description
POLLIN	x	x	Normal or priority data can be read
POLRDNORM	x	x	Normal data can be read
POLLRDBAND	x	x	Priority (OOB) data may be read
POLLPRI	x	x	High-priority data may be read

`poll()` – Event flags(cont.)

Constant	events	revents	Description
POLLOUT	x	x	Normal data may be written
POLLWRNORM	x	x	Equivalent to POLLOUT
POLLWRBAND	x	x	Priority (OOB) data may be written
POLLERR		x	An error has occurred on socket
POLLHUP		x	The hangup state
POLLNVAL		x	Something was wrong with the socket descriptor *fd*

Example

```
struct pollfd ufds[2];
s1 = socket(AF_INET, SOCK_STREAM, 0);
s2 = socket(AF_INET, SOCK_STREAM, 0);
//connect to server…
ufds[0].fd = s1;
ufds[0].events = POLLIN;
ufds[1].fd = s2;
ufds[1].events = POLLOUT;
rv = poll(ufds, 2, 3500);
if (rv == -1) {
    perror("poll"); // error occurred in poll()
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 3.5 seconds.\n");
} else {
    // check for events on s1:
    if (ufds[0].revents & POLLIN)
        recv(s1, buf1, sizeof buf1, 0);

    // check for events on s2:
    if (ufds[1].revents & POLLOUT)
        send(s2, buf2, sizeof buf2, 0);
}
```

select vs poll

- They provides similar functionalities but
  - select has a better portability because of its long existence
  - poll hadn’t been supported until Windows Vista or broken implementation in Mac OS X 10.3
  - select uses 3-bit of data per file descriptor while poll uses 64-bit, so invoking call needs to copy more over kernel space
  - select can offer microsecond timeouts while poll can only provide milliseconds
  - poll handles more than 1024 file handles at the same time (select default)
  - poll offers more flavors of events to wait for, and to receive.
- In practice, however, it’s hard to see differences between these