I/O Multiplexing – select()

Lecture 5

Introduction – 1/3

- Our TCP server can handle two inputs at the same time
  - **Standard input**
  - **TCP socket**
- Encountered a problem when a client was blocked in a call to **fgets()** and server process was killed
- The server TCP sends a FIN to the client TCP, but since client process was blocked reading from standard i/p, it never saw the EOF until it read from socket

Introduction – 2/3

- We need the capability to tell the kernel that we want to be notified if I/O conditions are ready (i.e. input is ready to be read, or descriptor is capable of taking more o/p)
- This capability is called **I/O multiplexing**
- It is provided by **select()** and **poll()** functions

Introduction – 3/3

- I/O multiplexing is used in networking applications in the following
  - When client is handling multiple descriptors
  - When client handles multiple sockets at the same time
  - If TCP server handles both listening socket and connected sockets
  - If server handles both TCP and UDP
  - If server handles multiple services and multiple protocols

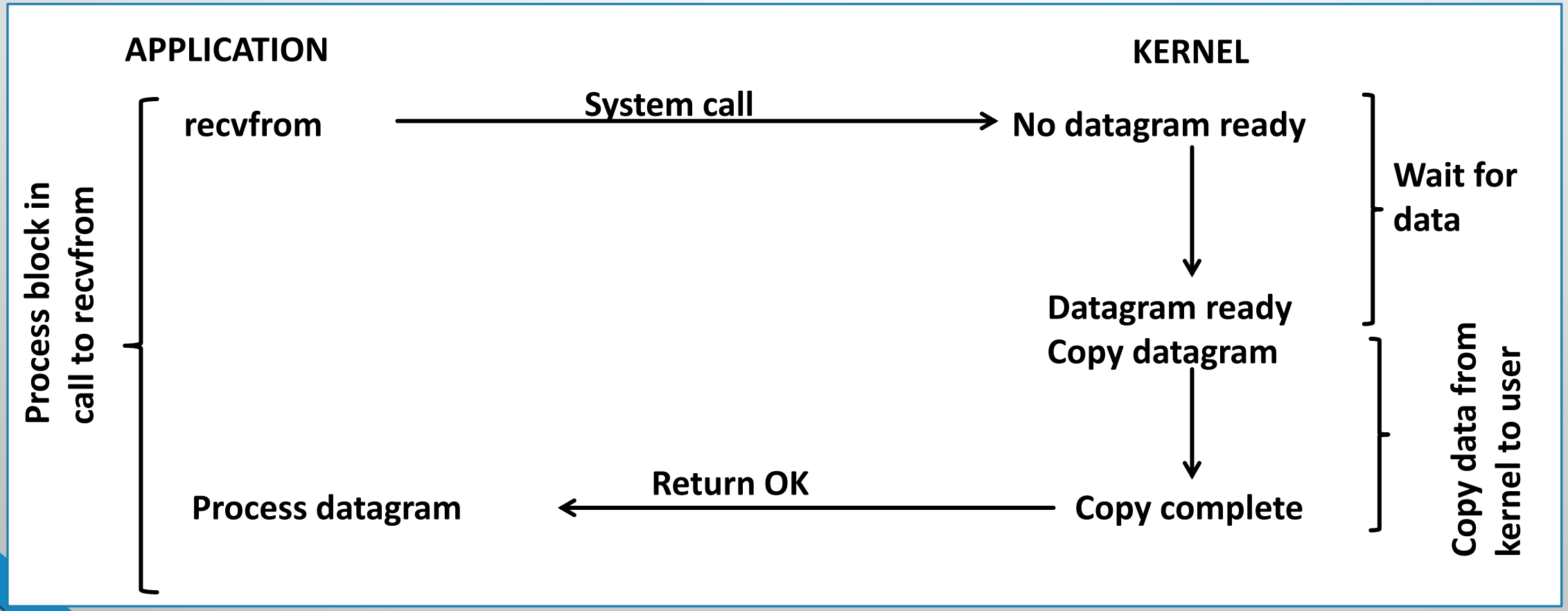
I/O Models

- Some of the I/O models are
  - Blocking I/O model
  - Non-Blocking I/O model
  - I/O multiplexing model
- There are normally two distinct phases for input operation
  - Waiting for data to be ready
  - Copying data from kernel to process

Blocking I/O model – 1/2

- Most prevalent model
- **By default, all sockets are blocking**
- The process calls recvfrom() and call does not return until datagram arrives and is copied into the application buffer, or error occurs
- The process is blocked the entire time from when it calls recvfrom() until it returns
- When recvfrom() returns successfully, the application processes the datagram

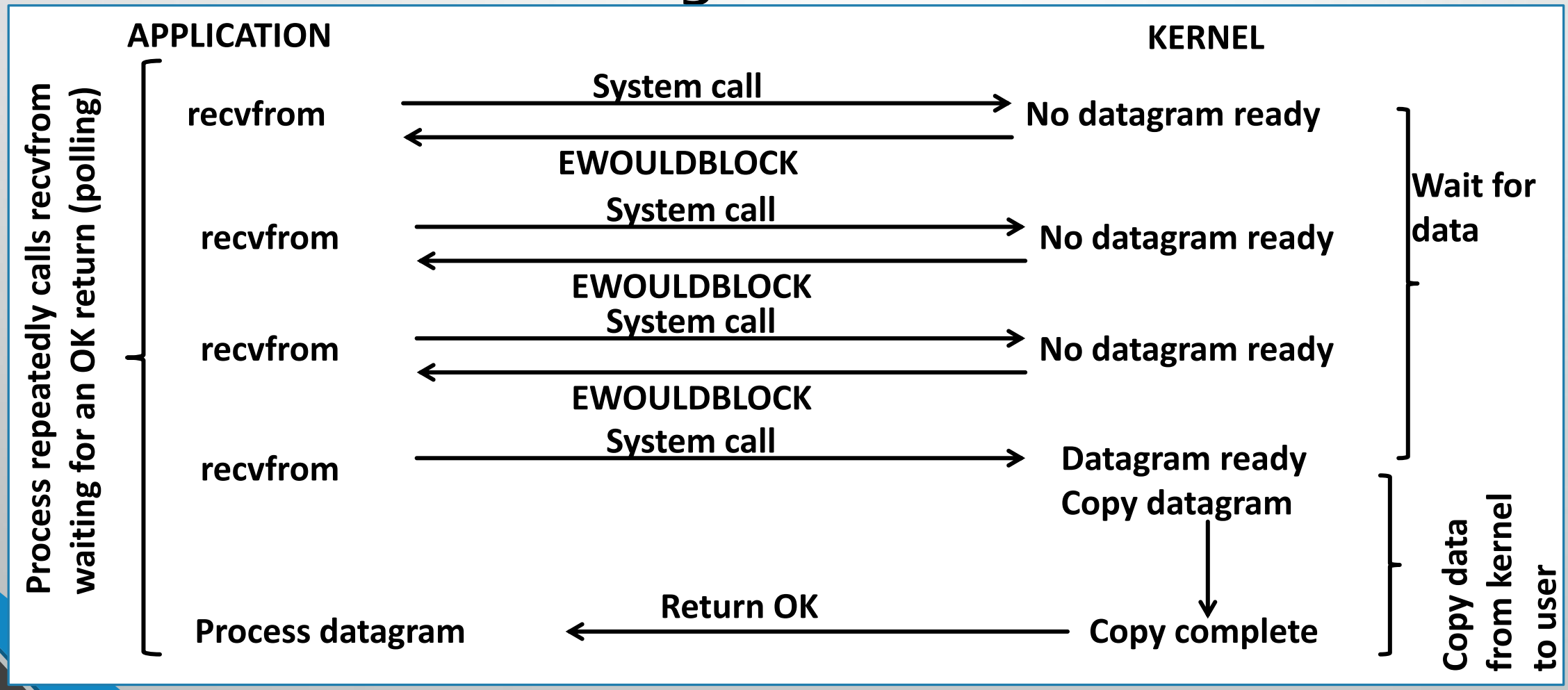
Blocking I/O model – 2/2



Non-Blocking I/O model – 1/2

- Here, we are telling the kernel that when I/O operation that is requested cannot be completed without putting the process to sleep, then do not put the process to sleep, but return an error instead
- The kernel will return **EWOULDBLOCK** instead of waiting
- When an application sits in a loop calling recvfrom() on a non-blocking descriptor, it is called **polling**

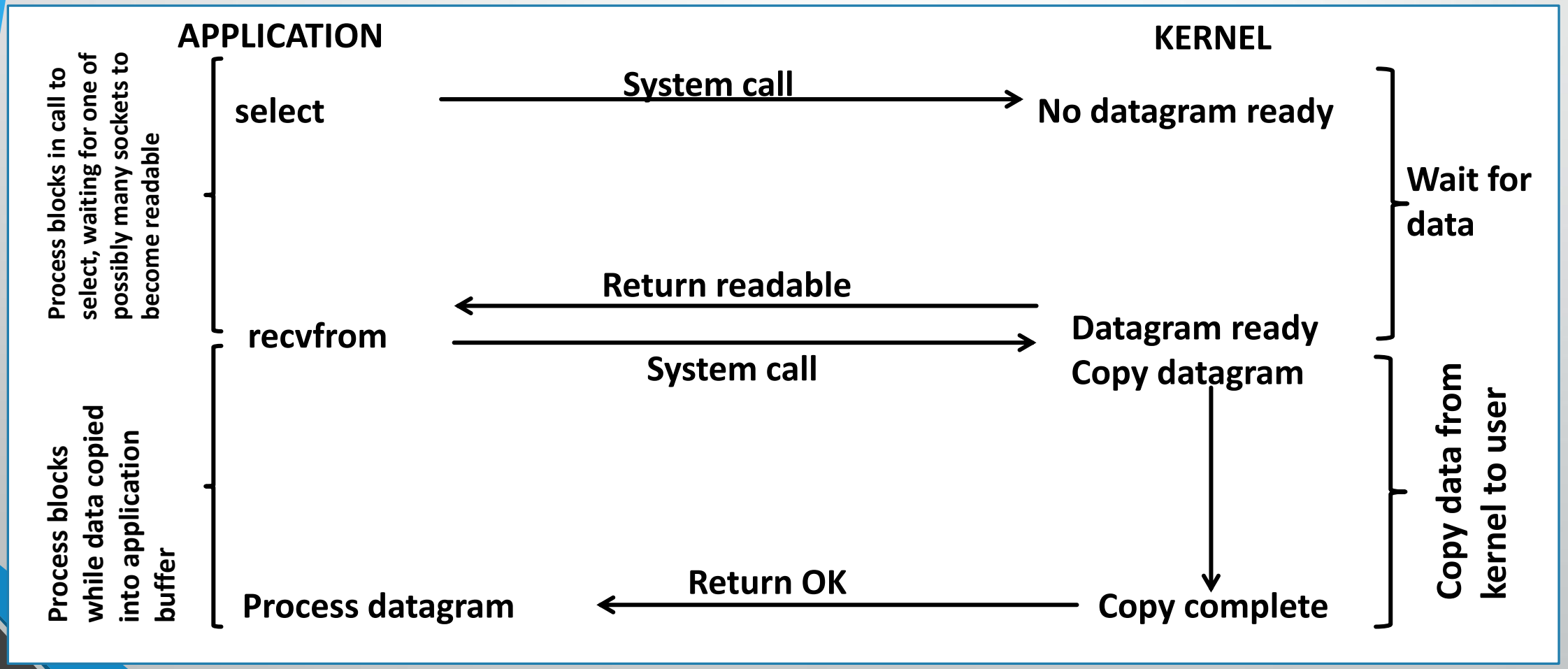
Non-Blocking I/O model – 2/2



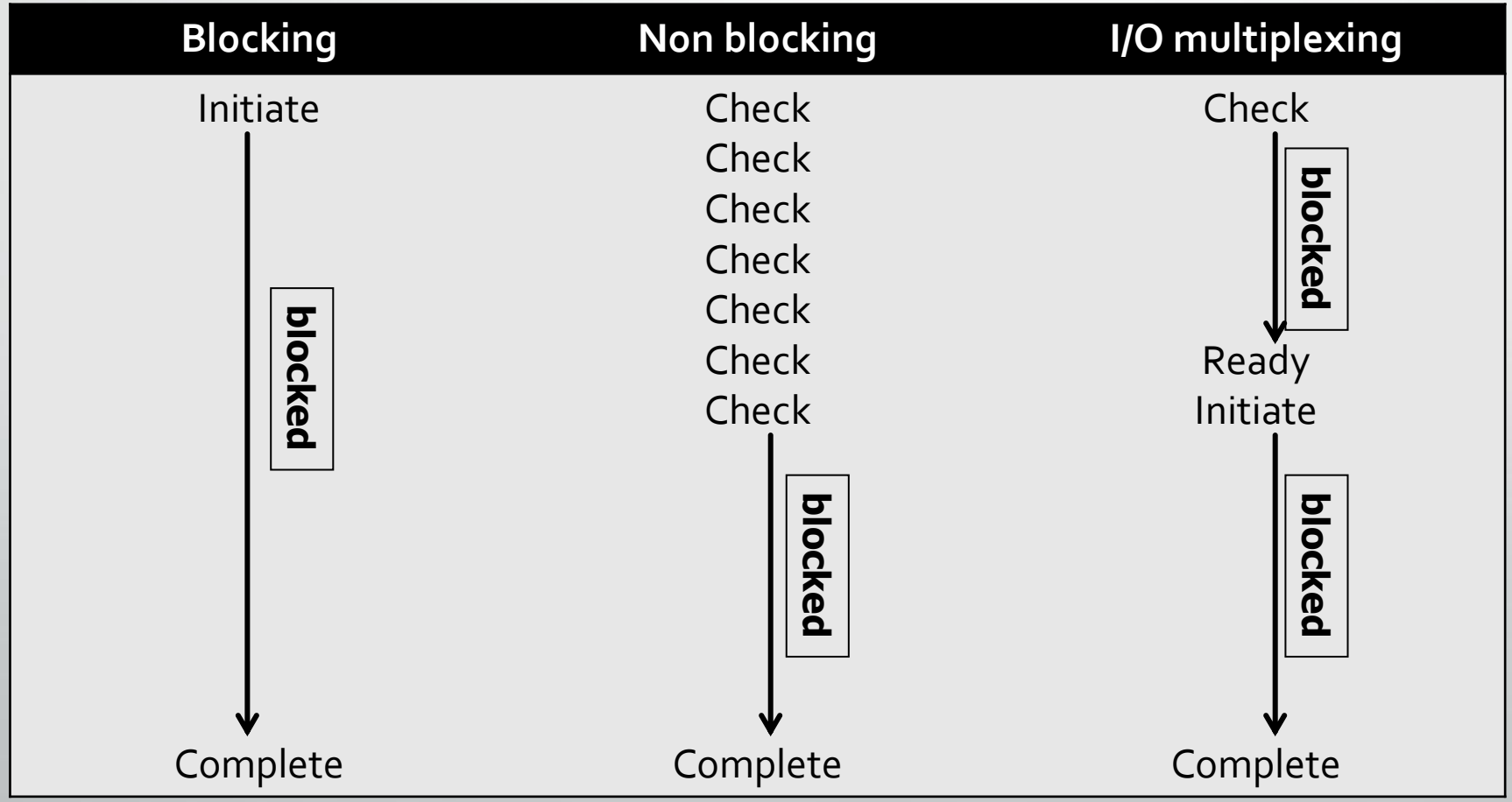
I/O multiplexing model – 1/2

- Call select() or poll() and block in one of these two system calls instead of blocking the actual I/O system call
- Block in call to select(), waiting for datagram to be readable
- When it is readable, call recvfrom() and process datagram

I/O multiplexing model – 2/2



Comparison of the I/O models



select() – 1/8

- **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**
- We tell the kernel what descriptors we are interested in (for read, write, exception handling)
- Descriptors are not restricted to sockets

select() – 2/8

- Syntax

  #include <sys/select.h>

  #include <sys/time.h>

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

- Return value
  - Positive count of ready descriptors
  - 0 on timeout
  - -1 on error

select() – 3/8

- Three possibilities
  - **Wait forever**
    - Return only when one of the specified descriptors is ready for I/O
    - Specify *timeout* argument as a **null pointer**
  - **Wait up to 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 timeval structure pointed to by *timeout* argument

select() – 4/8

- **Do not wait at all**
  - Return immediately after checking the descriptors
  - This is called **polling**
  - The **timeout argument** must point to a timeval structure and timer value must be o
- The *const* qualifier on *timeout* argument means it is not modified by select on return

select() – 5/8

- Eg. If we specify a time limit of 10 secs and select returns before the timer expires with one or more descriptors ready or with an error EINTR, the timeval structure is not updated with the number of seconds remaining when the function returns

- To specify one or more descriptor values for each of the three arguments : readset, writeset and exceptset, select uses descriptor sets, typically array of integers, with each bit corresponding to a descriptor

select() – 6/8

All implementation details are irrelevant to the application and are hidden in the **fd_set** datatype and following 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_CLEAR (int fd, fd_set *fdset);          //turn off bit for fd in fdset

int FD_ISSET (int fd, fd_set *fdset);          //Is the bit for fd on fdset?
```

select() – 7/8

Examples

```
fd_set rset;
FD_ZERO (&rset);                    //initialise the set : all bits off

FD_SET (1, &rset);                  //turn on bit for fd 1

FD_SET (5, &rset);                  //turn on bit for fd 5
```

select() – 8/8

- Any of the three arguments can be null if we are not interested
- The ***maxfd*** specifies the number of descriptors to be tested
- **Its value is the maximum descriptor to be tested plus one**
- The constant FD_SETSIZE defined in <sys/select.h> is the number of descriptors in fd_set datatype
- **Its value is often 1024**

select() - server program – 1/11

- **No need to use fork() to handle many clients**
- Server maintains only a **read descriptor set**
- The **descriptors 0,1 and 2 are set to STDIN, STDOUT and STDERR respectively**
- Therefore, the first available descriptor for **listening socket is 3**
- An array named **client** [ ] containing socket descriptor for each client is maintained
- All elements in the array are **initialized to -1(indicating available entry)**

select() - server program – 2/11

Fig 1: TCP server before the first client has established a connection

select() - server program – 3/11

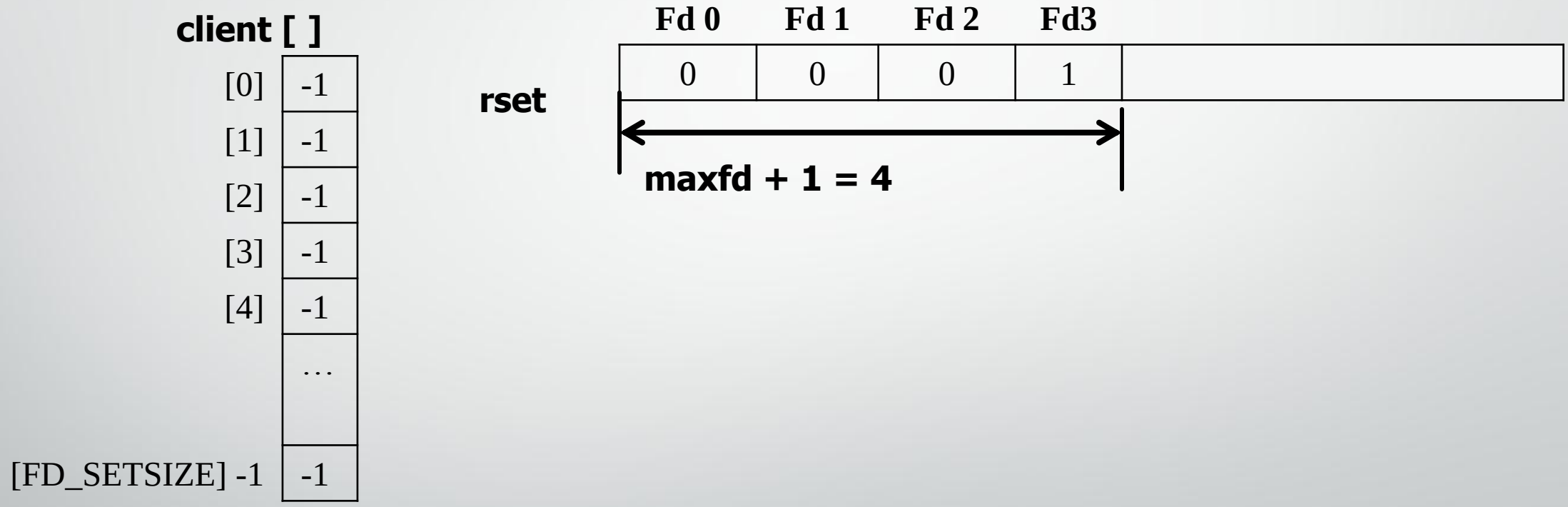


Fig 2: Data structures for TCP server with just a listening socket

select() - server program – 4/11

- When the first client establishes a connection with the server, the listening descriptor becomes readable and the server calls accept()
- **New connected descriptor returned by accept() will be 4**
- Server must remember the new connected socket in its client array, and connected must be added to the descriptor set

select() - server program – 5/11

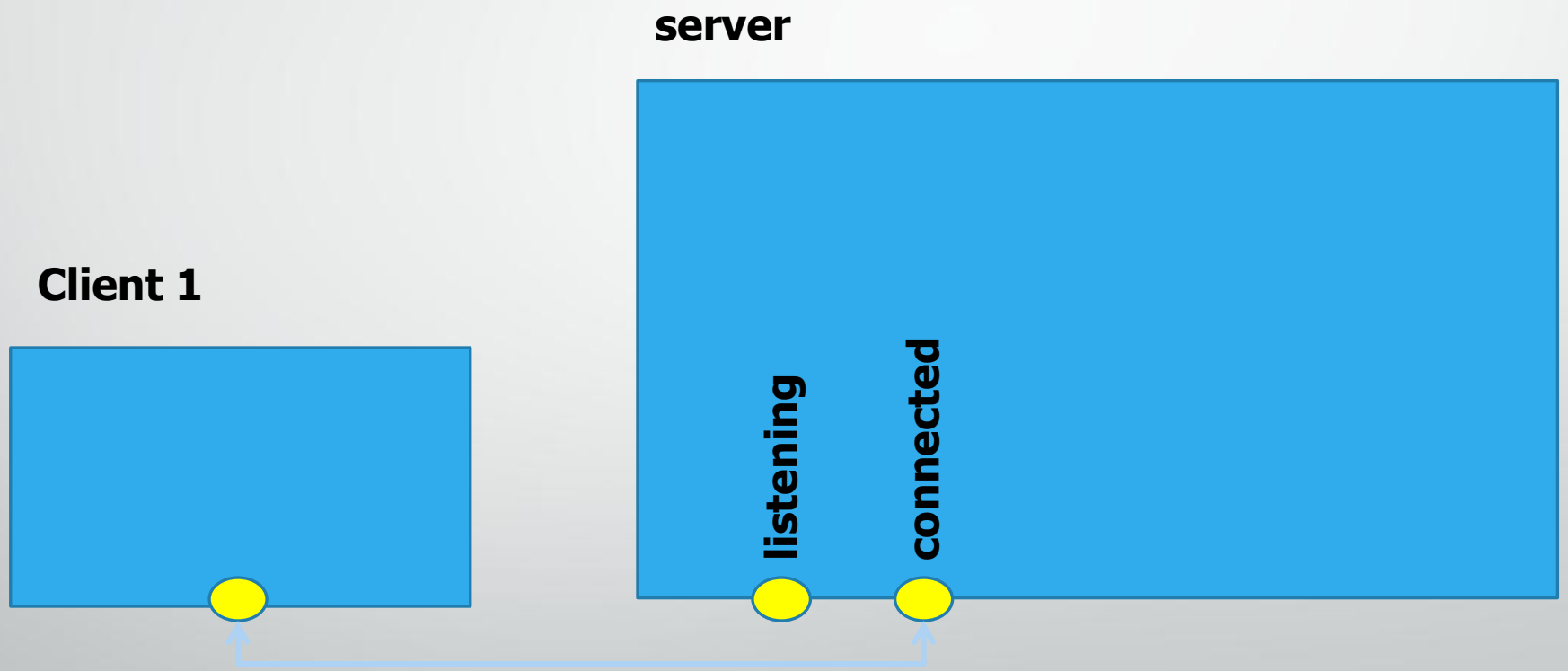


Fig 3 : TCP server after first client establishes connection

select() - server program – 6/11

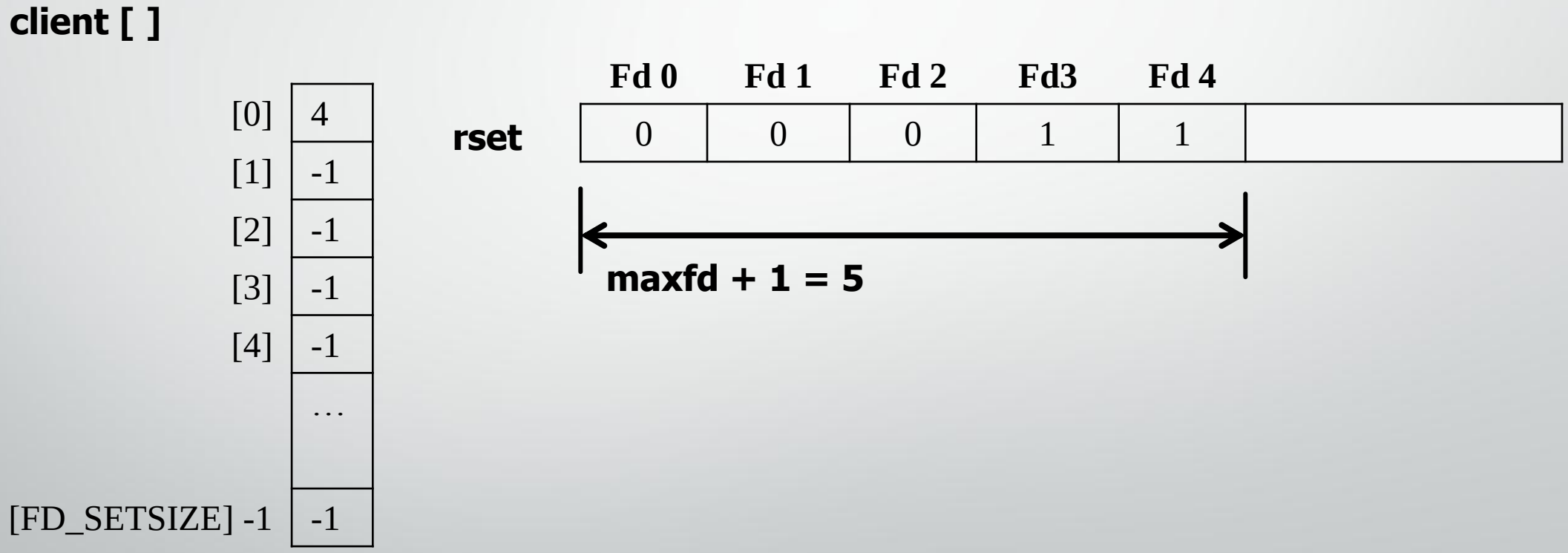


Fig 4: Data structures for TCP server after first client connected

select() - server program – 7/11

- Sometime later a second client establishes a connection

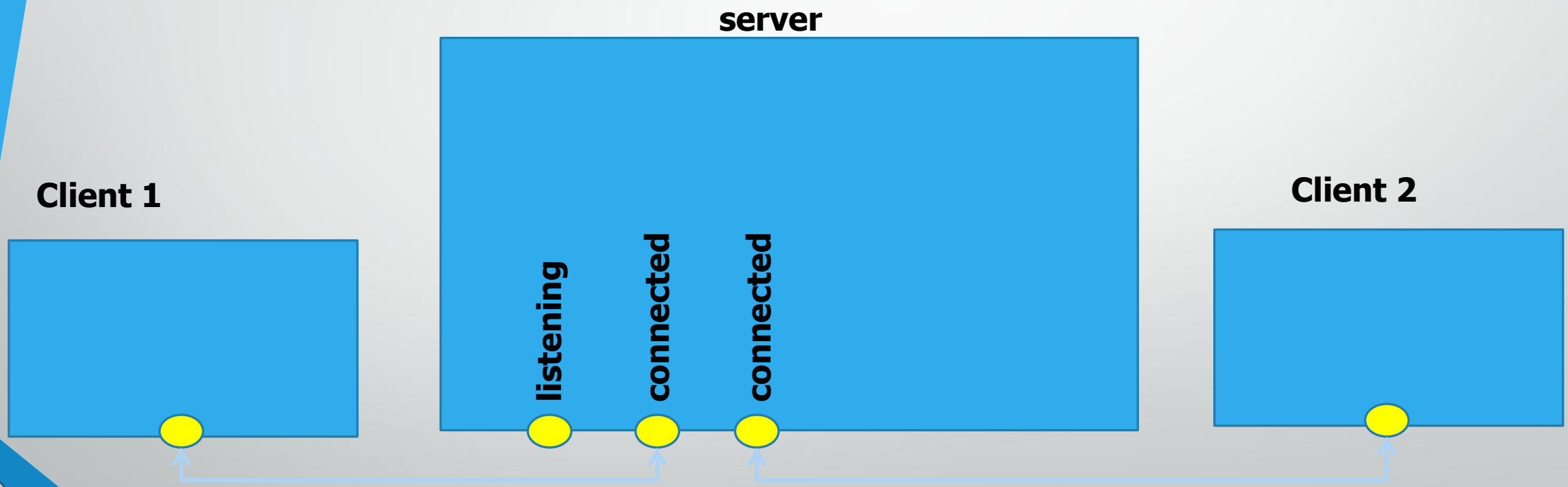


Fig 5 : TCP server after second client establishes connection

select() - server program – 8/11

- The new connected socket must also be remembered

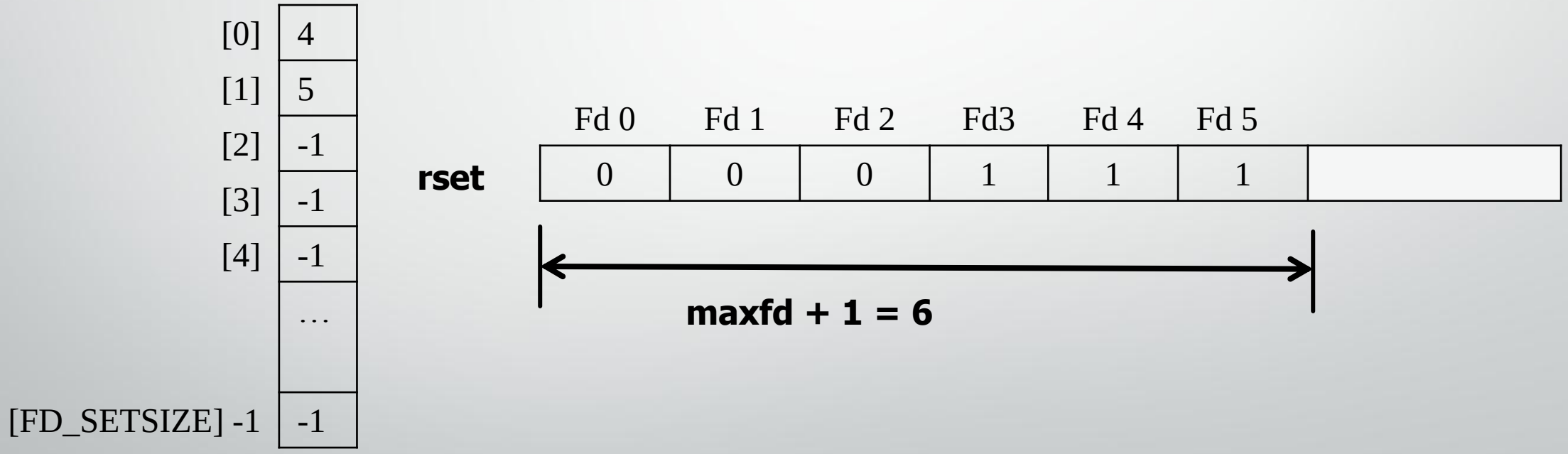


Fig 6: Data structures for TCP server after second client connected

select() - server program – 9/11

- Assume that the first client terminates connection, this makes descriptor 4 in server readable
- The client TCP sends a FIN, making descriptor 4 in the server readable
- When the server reads this connected socket, read() returns 0
- Value of client[0] is set to -1, and descriptor 4 in the set is set to 0

select() - server program – 10/11

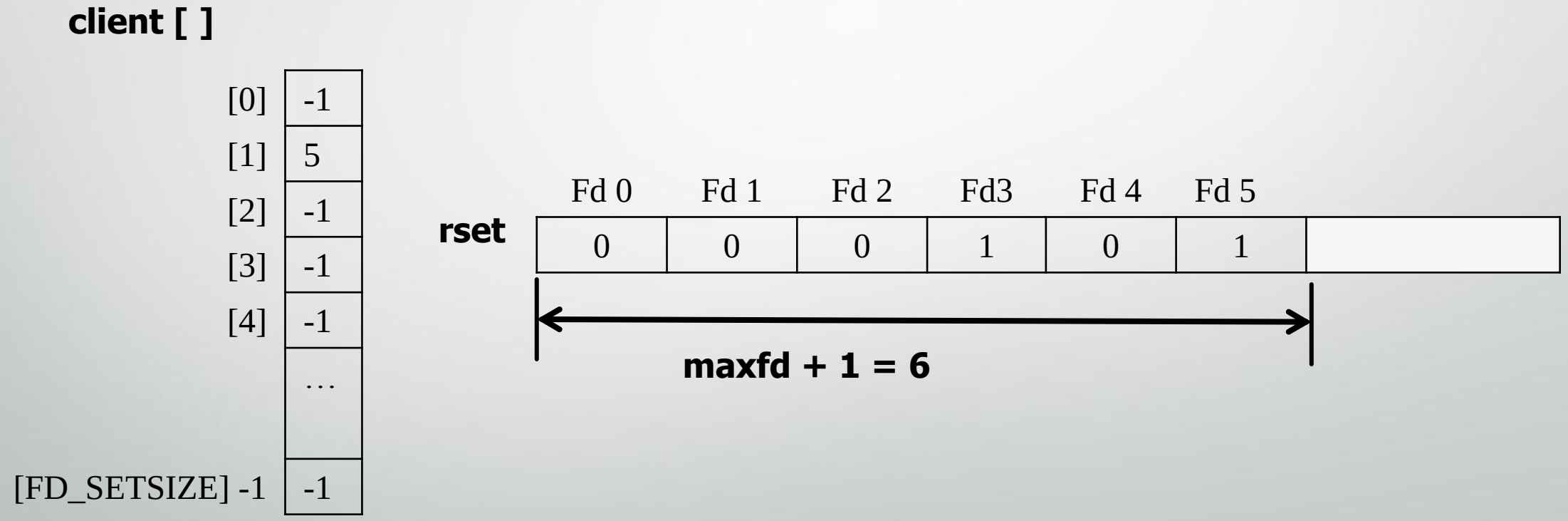


Fig 7: Data structures after first client terminates

select() - server program – 11/11

- As clients arrive, we record their connected socket descriptor in the first available entry in client[] array
- Also add connected socket to read descriptor set
- *maxi* is the highest index in the client[] array currently in use
- *maxfd (plus one)* is current value of the first argument to select()

Sample Program