

Module 3.2 – Blocking I/O and TCP multiplexing

Module 3 required reading material:

- [1] Brian “Beej Jorgensen” Hall, “Beej’s Guide to Network Programming, v3.1.11”. April 2023. <https://beej.us/guide/bgnet/html/split/>
 - Read chapter 7. I always start by reading Beej’s explanation since he’s straight to the point, but the level of details is lacking compared to the other two books.
- [2] Lewis Van Winkle, “Hands-On Network Programming with C”. Packt Publishing. May 2019. ISBN: 9781789349863. <https://learning.oreilly.com/library/view/hands-on-network-programming/9781789349863/>
 - Read chapter 3.
- [4] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, “The Sockets Networking API: UNIX® Network Programming Volume 1, Third Edition”. Addison Wesley. November 2003. ISBN: 0-13-141155-1. <https://learning.oreilly.com/library/view/the-sockets-networking/0131411551/>
 - This book provides a more in-depth/technical explanation for the topics covered in this module.
 - Read Chapter 6 – I/O Multiplexing. Read the entire chapter

In this module we’ll cover blocking I/O and TCP multiplexing. I’ll share my notes on the topics, but you need to read the material to get the most out of it.

Topics covered:

- Blocking
- Multiplexing
- Poll() and epoll()
- Select()

Blocking I/O

What is meant by blocking computer programming / Operating Systems?

Blocking refers to a situation in which a program or a specific operation halt or pauses the execution of the program until a certain condition is met or a task is completed. During this time, the program may appear unresponsive because it's waiting (sleeping) for something to happen.

Blocking can occur in various scenarios, such as:

- Blocking I/O: When a program performs input/output operations (like reading from a file or waiting for user input), it may block the execution until the operation is completed.
- Blocking Synchronization: When multiple threads or processes need to coordinate their work, they may block each other while waiting for certain resources to become available.

- **Blocking in Network Communication:** In networking, when a program sends or receives data over the network, it may block until the data transfer is finished or a timeout occurs.

Blocking can be problematic in certain situations, especially in applications that require responsiveness. In these cases, developers often use techniques like asynchronous programming or multithreading to avoid blocking and keep the application responsive.

Multiplexing TCP connections

We cover different ways to multiplex socket connections.

The socket APIs block by default. This is true whenever you use one of the following functions.

- `Accept()` blocks while waiting for incoming connection / until a new connection is available.
- `Recv()` blocks while waiting for new data to become available for reading.

Blocking can be problematic in certain situations, especially in applications that require responsiveness. For example,

- Our server application would only be able to serve one client / connection at a time.
- Servers (TCP/UDP, HTTP) serving more than one client cannot block while only one client is connecting/sending/receiving data.
- Applications usually need to serve multiple clients at a time/manage several connections simultaneously.

The same applies to client applications. A browser, for example, needs to be able to handle multiple connections (multiple tabs) to different websites, or a single web browser needs to be able to perform multiple actions when connecting to a server (downloading files, loading images, scripts, etc.)

Polling non-blocking sockets

Polling in computer programming refers to a technique where a program or process periodically checks the status or condition of a resource, data, or event to determine whether it has changed or meets a certain condition. This technique is often used when a program needs to monitor external events or resources asynchronously.

We can configure sockets to use non-blocking operations.

- In Linux: use `fcntl()` with the `O_NONBLOCK` flag.
- In Windows: use `ioctlsocket()` with the `FIONOBIO` flag.

Polling can be a waste of computer resources since most of the time there will be no data to read and adds to the program's complexity.

How is poll() different than polling? Oddly enough, the function poll() does the opposite of polling.

poll() takes an array of struct pollfds with information about which socket descriptors we want to monitor for and the OS blocks on poll() until one of those events occurs. For further information on poll() read the text and the man pages.

Multiplexing with the select() function.

We're going to focus on the select() function to learn about multiplexing. The concepts apply to poll() and epoll() as well. Research epoll() since that is the newest and the most efficient function out of the three and is used in modern applications, although not portable since it is Linux specific. There are connection limit issues with both poll() and select() which affect large applications, epoll() overcomes those issues. Older implementations of poll() and epoll() were not portable between Windows and Brekeley sockets making select() the preferred choice.

How does Multiplexing work?

In the context of networking and I/O operations, multiplexing refers to the technique of managing multiple input/output channels (such as network connections or file descriptors) concurrently within a program. The primary goal is to efficiently handle multiple communication tasks without the need for creating a separate thread or process for each task, which could be resource-intensive and less scalable.

In a nutshell, we're going to ask the operating system to do all the dirty work for us, and just let us know when some data is ready to read on which sockets. In the meantime, our process can go to sleep, saving system resources.

Read the textbooks, I'll provide my notes on select() below.

The select() function

select() gives you the power to monitor several sockets at the same time. select notifies the application which sockets are ready for reading, writing, and those which have raised exceptions. In a typical application, accept(), send(), and receive() are polling (monitoring) the socket waiting for data or connections. When using select(), we pass control to the kernel to do that work for us which allows our application to perform other work.

Key points:

- select() is slow when needing to handle giant numbers of connections but it is portable. You should look into libevent for a more efficient library.
- Given a set of sockets, it can be used to block until any of the sockets in that set is ready to be read from (written to?)
- Can be configured to return after a "time-out" period if none of the monitored events take place. (see the below)

Function declaration:

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

- Purpose
 - This function monitors "sets" of file descriptors; in particular: readfds, writefds, exceptfds. For read, write, and exceptions, respectively.

We'll cover two of the parameters in more detail below.

numfds

The `numfds` argument specifies the range of descriptors to be tested.

`numfds` should be set to the values of the highest file descriptor plus one. Why?

The reason for specifying `numfds` as the highest file descriptor plus one is related to the way the `select()` function's implementation handles bitsets (`fd_set` structures) internally.

Bitsets: Internally, `select()` uses bitsets to represent file descriptors. Each bit corresponds to a file descriptor, and the `select()` function checks the bits to determine which file descriptors have events pending. By specifying `numfds` as one greater than the highest file descriptor, it ensures that the bitset is large enough to represent all file descriptors you intend to monitor.

Readfds

If the `readfds` argument is not a null pointer, it points to an object of type `fd_set` that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

When `select` returns, `readfds` will be modified to reflect which of the file descriptors you select is ready for reading.

writefds

If the `writefds` argument is not a null pointer, it points to an object of type `fd_set` that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.

errorfds

If the `errorfds` argument is not a null pointer, it points to an object of type `fd_set` that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.

Return Value

- On success, `select()` and `pselect()` return the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in `readfds`, `writefds`, `exceptfds`). The return value may be zero if the timeout expired before any file descriptors became ready.
- On error, -1 is returned, and `errno` is set to indicate the error; the file descriptor sets are unmodified, and timeout becomes undefined.

Notes:

Assuming we want to monitor `readfds`, when `select` returns, `readfds` will be modified to reflect which of the file descriptors you select is ready for reading.

We can use `FD_ISSET()` to test them.

After `select()` returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions.

On success, `select()` itself returns the number of socket descriptors contained in the (up to) three descriptor sets it monitored. The return value is zero if it timed out before any sockets were readable/writable/excepted. `select()` returns -1 to indicate an error.

Each of these sets is of type `fd_set`.

Any or all of these parameters can be NULL if you're not interested in those types of events.

The first parameter, `numfds` is the highest-numbered socket descriptor (they're just ints, remember?) plus one.

```
s1 = socket(...);
s2 = socket(...);
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);
```

Since we got `s2` second, it's the "greater", so we use that to indicate the `numfds` param in `select()`.

```
n = s2+1
select( n, ...);
```

Macros to create, set, test, and clear `FD_SET`.

Function	Description
<code>FD_SET(int fd, fd_set *set);</code>	Add fd to the set.
<code>FD_CLR(int fd, fd_set *set);</code>	Remove fd from the set.
<code>FD_ISSET(int fd, fd_set *set);</code>	Return true if fd is in the set.
<code>FD_ZERO(fd_set *set);</code>	Clear all entries from the set.

struct timeval:

A time structure that allows specifying a time-out period for select/fd_sets.

Select returns if time-out period exceeds before any file descriptor is ready.

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

_micro_seconds, not milliseconds. There are 1,000 microseconds in a millisecond, and 1,000 milliseconds in a second. Thus, there are 1,000,000 microseconds in a second.

This causes select() to wait 1.5 seconds:

```
tv_sec = 1;
tv_usec = 500000;
```

Other things of interest:

If you set the fields in your struct timeval to 0, select() will timeout immediately, effectively polling all the file descriptors in your sets.

If you set the parameter timeout to NULL, it will never timeout, and will wait until the first file descriptor is ready.

Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to select().

One last thing to note about select(): if you have a socket that is listen()ing, you can check to see if there is a new connection by putting that socket's file descriptor in the readfds set.

Using the select() function:

```
1. Must first add our sockets into an fd_set.
   SOCKET socket_listen, socket_a, socket_b;

   //declare an fd_set variable
   fd_set our_sockets;
   FD_ZERO(&our_sockets);           //it's important
to zero-out the fd_set
```

```

        //Note: *sockets are added to a fd_set*
        FD_SET(socket_listen, &our_sockets);           //add
socket_listen to the our_socket set.
        FD_SET(socket_a, &our_sockets);
        FD_SET(socket_b, &our_sockets);

        //select() also requires that we pass a number that's larger than
the largest socket descriptor we are going to monitor.
        SOCKET max_socket;
        max_socket = socket_listen;
        if (socket_a > max_socket) max_socket = socket_a;
        if (socket_b > max_socket) max_socket = socket_b;

```

When we call `select()`, it modified our `fd_set` of sockets to indicate which sockets are ready.

It is recommended to make a copy of our socket set before calling `select()`.

```

        fd_set copy;
        copy = our_sockets;           //do they point to the same or are
they different objects?

```

```

        //the select call blocks until at least one of the sockets is
ready to be read from.
        //Only copy is modified when select() returns and only contains
sockets that are ready to be read from.
        //the argument position for the given FD determines what select
will monitor the sockets for :read, write, or exceptions.
        select(max_socket+1, &copy, 0, 0, 0);

        //checks if socket_listen is in the set of ready FDs copy.
        if (FD_ISSET(socket_listen, &copy))
            //perform action.

```

Iterating through an `FD_SET`:

Can use a simple `for` loop but must start at 1 not 0.

For each possible socket descriptor, we simply use `FD_ISSET()` to check `if` the socket descriptor is in the set.

```

        SOCKET i;
        for (i = 1; <= max_socket;i++)
        {
            //if (FD_ISSET(socket_descriptor_id, FD_SET))
            if (FD_ISSET(i, &master))
            {
                //perform action
            }
        }

```

```

Module 3.1 - TCP Sockets in-depth | TCP Multiplexing in-depth | Using the select() function, Unlabeled
1 Using the select() function:
2 1. Must first add our sockets into an fd_set.
3 SOCKET socket_listen, socket_a, socket_b;
4
5 //declare an fd_set variable
6 fd_set our_sockets;
7 FD_ZERO(&our_sockets); //it's important to zero-out the fd_set
8
9 //Note: "sockets are added to a fd_set"
10 FD_SET(socket_listen, &our_sockets); //add socket_listen to the our_socket set.
11 FD_SET(socket_a, &our_sockets);
12 FD_SET(socket_b, &our_sockets);
13
14 //select() also requires that we pass a number that's larger than the largest socket descriptor we are going to monitor.
15 SOCKET max_socket;
16 max_socket = socket_listen;
17 if (socket_a > max_socket) max_socket = socket_a;
18 if (socket_b > max_socket) max_socket = socket_b;
19
20 When we call select(), it modified our fd_set of sockets to indicate which sockets are ready.
21
22 It is recommended to make a copy of our socket set before calling select().
23 fd_set copy;
24 copy = our_sockets; //do they point to the same or are they different objects?
25
26 //the select call blocks until at least one of the sockets is ready to be read from.
27 //Only copy is modified when select() returns and only contains sockets that are ready to be read from.
28 //the argument position for the given FD determines what select will monitor the sockets for (read, write, or exceptions.
29 select(max_socket+1, &copy, 0, 0, 0);
30
31 //checks if socket_listen is in the set of ready FDs copy.
32 if (FD_ISSET(socket_listen, &copy))
33     //perform action.
34
35 Iterating through an FD_SET:
36 Can use a simple for loop but must start at 1 not 0.
37 For each possible socket descriptor, we simply use FD_ISSET() to check if the socket descriptor is in the set.
38
39 SOCKET i;
40 for (i = 1; i <= max_socket; i++)
41 {
42     //if (FD_ISSET(socket_descriptor_id, FD_SET))
43     if (FD_ISSET(i, &master))
44     {
45         //perform action
46     }
47 }

```

Poll() vs Select()

- Select()
 - It's part of the POSIX standard, it is more portable across different operating systems.
 - Has a limit on the number of file descriptors it can handle with a default of 1024.
- Poll()
 - May be more efficient for large-scale applications.
 - Doesn't have a limit on the number of file descriptors it can handle.
- In modern programming, you might use more advanced mechanisms like epoll(). Not covered here since it is not portable.

Code + assignments:


- From the network_programming directory, go through the tcpserver_multiplex.c file and read my comments.

```

$ pwd
/home/kali/network_programming/code/src/tcp_server

(kali@kali)-[~/network_programming/code/src/tcp_server]
$ ll
total 20
-rw-r--r-- 1 kali kali 6407 Nov  4 21:50 server.c
-rw-r--r-- 1 kali kali 3270 Nov  4 21:50 server_loop.c
-rw-r--r-- 1 kali kali 4818 Nov  4 21:50 tcpserver_multiplex.c

```



From the netlib.h file, read the select_loop() and udp_select_loop() functions

```

C netlib.h x
C netlib.h > select_loop(int)
293
294 > void strip_new_line(char *input) ...
307
308 //returns 1 on success, 0 on failure.
309 > int send_recv_loop(int socket_peer) ...
375
376
377 int select_loop(int sockfd)
378 {
379     TRACE_ENTER();
380

```

```

9
10 > /* ...
17 > /*void make_udp_addrinfo_ipv4(struct addrinfo *addr_struct, int
19
20 int udp_select_loop(int socket_listen)
21 {
22     TRACE_ENTER();
23     int ret_val = 1;    //1 == true;
24     int READ_SIZE = 1024;
25
26     fd_set master;
27     FD_ZERO(&master);
28     FD_SET(socket_listen, &master);
29     int max_socket = socket_listen;
30     printf("Waiting for connections ...\n");
31
32     while(1)

```

- Rewrite the programs in "Hands-On Network Programming with C" chapter 3 to work on Linux only.

- Write a TCP client and server program.
- Add select() to the previous program.
- Add the ability to handle sending and receiving partial data.
- Research the epoll() function.
 - <https://man7.org/linux/man-pages/man7/epoll.7.html>
 - <https://linux.die.net/man/7/epoll>