- send
- recv
- bind
- connect
- inet_addr
- setsockopt
- getsockname
- fcntl

  fcntl - manipulate file descriptor

  ```
  #include <fcntl.h>
  int fcntl(intfd, intcmd, ... /*arg*/ );
  ```

  fcntl(fd, F_SETFL, O_NONBLOCK);

How to Build HTTP server

Prerequisite

- Learn about what is **OSI**.

  the OSI provides a standard for different computer systems to be able to communicate with each other. The OSI Model can be seen as a universal language for computer networking. It's based on the concept of splitting up a communication system into seven abstract layers, each one stacked upon the last.

- Learn about OSI's 4th Layer : **Transport Layer**.

  - The Transport layer provides flow control and error handling, and participates in solving problems concerned with the transmission and reception of packets.
  - Common examples of Transport layer protocols are Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Sequenced Packet Exchange (SPX).
  - In short, from RFC 2616:
    HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80, but other ports can be used. HTTP presumes a reliable transport.

- Learn **TCP** socket programming.

  - What is **socket**?

    Sockets allow communication between two different processes on the same or different machines. It's a way to talk to other computers using standard Unix file descriptors.

Socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

▼ Basic notion TCP/IP sockets.

▼ Create a socket.

```
int server_fd = socket(domain, type, protocol);
```

- domain or address family :
  - communication domain in which the socket should be created.
  - For TCP/IP sockets, we want to specify the IP address family "AF_INET".
- type :
  - type of service.
  - For TCP/IP sockets, we want to specify the virtual circuit service "SOCK_STREAM".
- protocol :
  - indicate a specific protocol to use in supporting the sockets operation.
  - Since there's only one form of virtual circuit service, there are no variations of the protocol, so the last argument, protocol, is zero.

```
#include <sys/socket.h>
...
...
if ((server_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        std::cerr << "error: socket creation" << std::endl;
        exit(1);
    }
```

▼ Identify a socket.

- When we talk about naming a socket, we are talking about assigning a transport address to the socket (a port number in IP networking). In sockets, this operation is called binding an address and the bind system call is used for this.

```
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

- socket :
  - The first parameter, socket, is the socket that was created with the socket system call.
- sockaddr

- For the second parameter, the structure sockaddr is a generic container that just allows the OS to be able to read the first couple of bytes that identify the address family.
- For IP networking, we use struct sockaddr_in, which is defined in the header <netinet/in.h>

```
struct sockaddr_in
{
    __uint8_t         sin_len;
    sa_family_t       sin_family;
    in_port_t         sin_port;
    struct in_addr    sin_addr;
    char              sin_zero[8];
};
```

▼ **Before calling bind, we need to fill out this structure.**

- sin_family
  - The address family we used when we set up the socket.
  - In our case, it's "AF_INET".
- sin_port
  - The port number (the transport address). You can explicitly assign a transport address (port) or allow the operating system to assign one.
  - If you're a client and won't be receiving incoming connections, you'll usually just let the operating system pick any available port number by specifying port 0.
  - If you're a server, you'll generally pick a specific number since clients will need to know a port number to connect to.
- sin_addr
  - The address for this socket. This is just your machine's IP address.
  - Most of the time, we don't care to specify a specific interface and can let the operating system use whatever it wants.
  - The special address for this is 0.0.0.0, defined by the symbolic constant "INADDR_ANY".

- address_len
  - the third parameter specifies the length of that structure.
  - This is simply sizeof(struct sockaddr_in).

```
#include <sys/socket.h>
…
struct sockaddr_in address;
const int PORT = 8080; //Where the clients can reach at
/* htonl converts a long integer (e.g. address) to a network representation */
/* htons converts a short integer (e.g. port) to a network representation */
memset((char *)&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(PORT);
if ((bind(listen_fd, (struct sockaddr *) &address, sizeof(address))) < 0) {
        std::cerr << "error: socket creation" << std::endl;
        exit(1);
    }
```

▼ Server wait incoming connection.

- Before a client can connect to a server, the server should have a socket that is prepared to accept the connections.
  The listen system call tells a socket that it should be capable of accepting incoming connections.

```
#include <sys/socket.h>
int listen(int socket, int backlog);
```

- backlog
  - The second parameter, backlog, defines the maximum number of pending connections that can be queued up before connections are refused.
- The original socket that was set up for listening is used only for accepting connections, not for exchanging data.
- The accept system call grabs the first connection request on the queue of pending connections (set up in listen) and creates a new socket for that connection.

```
#include <sys/socket.h>
int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);
```

- socket
  - The first parameter, socket, is the socket that was set for accepting connections with listen.
- address
  - The second parameter, address, is the address structure that gets filed in with the address of the client that is doing the connect.
- address_len

    - The third parameter is filled in with the length of the address structure.

```
if (listen(server_fd, 3) < 0) {
    std::cerr << "error: listern call" << std::endl;
    exit(EXIT_FAILURE);
}
if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)addrlen))<0){
    std::cerr << "error: accept call" << std::endl;
    exit(EXIT_FAILURE);
}
```

- By default, socket operations are synchronous, or blocking, and accept will block until a connection is present on the queue.
- The answer to this is non-blocking I/O. We set a flag on a socket which marks that socket as non-blocking.
- this means that, when performing calls on that socket (such as read and write), if the call cannot complete, then instead it will fail with an error like EWOULDBLOCK or EAGAIN.
- To mark a socket as non-blocking, we use the fcntl system call.

```
//WARNING!!! DO NOT USE THIS IN WEBSERV 42 PROJECT
#include <fcntl.h>
int flags;
if ((flags = fcntl(server_fd, F_GETFL)) < 0){
    std::cerr << "could not get file flags" << std::endl;
    exit(1);
}
```

- F_GETFL
    - Return (as the function result) the file access mode and the file status flags; arg is ignored.

```
if (fcntl(server_fd, F_SETFL, flags | O_NONBLOCK) < 0){
    std::cerr << "could not set file flags" << std::endl;
    exit(1);
}
```

- F_SETFL
    - F_SETFL overwrites the flags with exactly what you pass in the third parameter, "flags" the old parameter, "|" then the new flag in, here O_NONBLOCK.

▼ Send and receive messages.

- Before to send and receive data a client need to connect to the socket

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

- This function connects your socket to the address specified in the sockaddr structure.
- Now to communicate with our server (send and receive data), we will use the send and recv functions.

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int socket, const void *buf, size_t len, int flags);
```

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int socket, void *buf, ssize_t len, int flags);
```

- buf
  - The data we want to send.
- len
  - The size of the data.
- flags
  - Int flags are equal to 0, it means that no flags are specified. these are optional.

```
int sock;
char buffer[1024];
[...]
if(send(sock, buffer, strlen(buffer), 0) < 0){
    std::cerr << "Could not send data" << std::endl;
    exit(errno);
}
```

- In this example, we are sending a string, but we could very well send something else like a structure or an int for example.

```
char buffer[1024];
int n = 0;

if((n = recv(sock, buffer, sizeof buffer - 1, 0)) < 0){
    std::cerr << "Could not recieve data" << std::endl;
    exit(errno);
}

buffer[n] = '\0';
```

- For the reception of data, we must make sure to place the final \0 in our string (hence the -1 in the recv).
  In the case of the reception of a structure, it would obviously not be necessary to put this -1.

▼ Close the socket.

- When we're done communicating, the easiest thing to do is to close a socket with the close system call; the same close that is used for files.

```
#include <unistd.h>
int close(int fd);
```

```
close(new_socket);
```

▼ Advanced sockets.

- The select function waits for a change of state of the descriptors contained in different sets.

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

- readfds
  - The descriptors will be monitored to see if read data is available, a call to recv for example, will therefore not block.
- writefds
  - The descriptors will be monitored for writing to see if there is space to write the data, so a call to write will not block.
- timeout
  - The timeout parameter allows you to place a time limit on waiting for select.
- When the state of one of the descriptors changes select returns a value> 0, and the sets are changed.
- It is then necessary to check the state of each of the descriptors of the sets via the macros presented below.

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

- FD_CLR
  - Removes the descriptor fd from the set.
- FD_ISSET
  - Checks if the descriptor fd is contained in the set after the call to select.
- FD_SET
  - Adds the descriptor fd to the set.
- FD_ZERO
  - Empty the set.

▼ Summary.

So what to use when and in what order?

- TCP client side
  - socket
  - connect
  - send
  - recv
  - close
- TCP server side
  - socket
  - fcntl
  - bind
  - listen
  - accept
  - send
  - recv
  - close

▼ Example.

- TCP socket client-side code:

https://github.com/Sherchryst/webserv/tree/main/Docs/POC/sgah/client

- TCP socket server-side code:

https://github.com/Sherchryst/webserv/tree/dev/Docs/POC/sgah/server

- Select function use

https://github.com/Sherchryst/webserv/tree/dev/Docs/POC/sgah/select

▼ Learn about CGI

▼ What is CGI?

- CGI Common Gateway Interface is an interface specification that enables web servers to execute an external program, typically to process user requests
- Instead of send the content of a file(HTML file, image), the HTTP server execute a software and return the generated content.
- CGI is the industrial standard who indicate how to send the request from the HTTP server to the sofware and how to collect the answer generated.
- A classic example of parameter is the string containing the terms searched for by a search engine.

▼ What it's used for?

A typical use case occurs when a Web user submits a Web form on a web page that uses CGI.

- The form's data is sent to the Web server within an HTTP request with a URL denoting a CGI script.
- The Web server then launches the CGI script in a new computer process, passing the form data to it.
- The output of the CGI script, usually in the form of HTML, is returned by the script to the Web server, and the server relays it back to the browser as its response to the browser's request

▼ CGI in web browsing.

▼ To understand the concept of CGI, let's see what happens when we click a hyperlink to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demand for the URL ie. filename.
- Web Server will parse the URL and will look for the filename. If it finds requested file then web server sends that file back to the browser otherwise sends an error message