BUILDING A WEB SERVER

Imp. Syscalls:

① read:

```
int read (
    int fd,        → file descriptor (0, 1, 2 or given by kernel)
    void *buf,     → memory addr of buffer where the value is stored or taken.
    size_t count   ← No. of characters or size count
)
```

② write:

```
int write (
    int fd,
    void *buf,
    size_t count
)
```

③ open:

```
int open (
    char *pathname,
    int flags,        ← tells whether read only, write only, read-write both, etc.
    mode_t mode
)
```

→ Entire Process block is stored or created in kernel memory using a blob of data or data structure called as `struct task_struct * current`

↳ represents process block of current process.

→ Network System calls:

① socket: → creates endpoint for communication & returns a file descriptor that refers to endpoint

```
int socket (
    int domain,
    int type,
    int protocol
)
```

Date: / /

② bind :- used to bind created socket and create a direct connection between processes/systems.

```
int bind (
    int sockfd,
    struct sockaddr *addr,   // current address of socket in kernel memory.
    socklen_t addrlen
)
```

③ struct sockaddr_in :-

```
struct sockaddr {
    uint16_t sa_family;     // 2 Byte representation
    uint8_t  sa_data[14];   // 1-byte representation.
};
```

④ struct sockaddr_in {

```
    uint16_t sin_family;    // 2-Byte
    uint16_t sin_port;      // 2-Byte
    uint32_t sin_addr;      // 4-Byte
    uint8_t  __pad[8];      // Padding of 8 byte.
};
```

} total 8 byte structure

eg:

```
struct sockaddr_in addr = {
    AF_INET,                // Default value
    02 00   htons(80),      // → http port 80.
    {inet_addr("127.0.0.1")},   // IP address for connectivity
    {0}                     // Padding with '0' for rest of remaining bits.
};
```

→ It is stored in 'BIG ENDIAN' FORMAT instead of Little Endian; i.e. from Most significant to least significant bits left to right eg '127.0.0.1' is '7f 00 00 01'

↓

stored not in reversed order; i.e. Not little endian.

⑤ listen() → used to listen for clients on server for connection.

```
int listen (
    int sockfd,
    int backlog.
    )
```

⑥ accept() → used to accept connection when found listening.

```
int accept (
    int sockfd,
    struct sockaddr.* addr,
    socklen_t *addrlen
    )
```

→ Understanding HTTP for multiple clients handling ie Multiprocessing (using FORK())

① Normal Single client (NO fork())

Protocol streaming

TCP IP fd

Socket (AF_INET, SOCK_STREAM, IPPROTO_IP) = 3

bind (3,
{ sa_family = AF_INET,
sin_port = htons(80),
sin_addr = inet_addr("0.0.0.0")},
16) = 0 → fd → file descriptor in kernel

listen (3, 0) = 0

accept (3, NULL, NULL) = 4

read (4,
"GET /flag HTTP/1.0\r\n\r\n",
256) . = 19

open ("/flag", O_RDONLY) = 5

read (5, "FLAG", 256) = 4

write (4,
"HTTP/1.0 200 OK\r\n\r\nFLAG",
27) = 27

close (4).

② Using fork for Multiple clients

Socket (AF_INET, SOCK_STREAM, IPPROTO_IP) = 3

bind (3,
{sa_family = AF_INET,
sin_port = htons(80),
sin_addr = inet_addr("0.0.0.0")},
16) = 0

listen (3, 0) = 0

accept (3, NULL, NULL) = 4

fork() = 43

close (4) = 0

accept (3, NULL, NULL) = 4

fork() = 44

:

SO... OS,

creates a child with PID = 43 which will now carry out further processing for req of the client with fd = 4 and parent will become free to listen.

fork() = 0

close (3) = 0

fork() = 0

--- operations ---

close (4)

→ Exit:

Now, we have to do exit syscall as we done before in module 1 & 2.

```
.intel_syntax noprefix
.global _start
.section .text
_start:
    mov rdi, 0
    mov rax, 60
    syscall
.section .data
```

same & new

→ Socket:

Now, let's create a socket for our webserver.

```
.intel_syntax noprefix
.global _start
_start:
    mov rax, 41      ; "41" is Socket_syscall no. and can be obtained by; grep -R "#define __NR_socket" /usr/include
    mov rdi, 2       ; '2' is AF_INET_VALUE and can be obtained by 'grep -R "AF_INET" /usr/include
    mov rsi, 1       ; '1' is SOCK_STREAM value and is obtained by 'grep -R "SOCK_STREAM" /usr/include
    mov rdx, 0       ; '0' is IPPROTO_IP and is usually '0' and if diff can be obtained by 'grep -R "IPPROTO_IP" /usr/include'
    syscall
    mov rdi, 0
    mov rax, 60
    syscall
```

for socket

for exit

all are defaults

→ Bind:

Now, we have to assign our socket a Network identity. For, this we will use bind syscall to connect our socket to specific IP address and port number.

The call requires us to provide, socket file descriptor, a pointer to struct sockaddr — specifically struct sockaddr_in for IPV4 that holds fields like addr family, port, & IP addr and size of that structure.

Binding is essential as it ensures our server listens on a known address, making it reachable by clients.

Now;

```
.intel_syntax noprefix
.global _start
.section .text
_start:
```

To call .data section into actual code for execution. Entire code should be written in this).

```
mov rax, 41          ← Syscall for socket or SOCK_SYS.
mov rdi, 2           ← Default AF_INET Value when creating socket.
mov rsi, 1           ← value for SOCK_STREAM or Socket stream Value '1' for TCP IP connection.
mov rdx, 0           ← Default IPPROTO_IP value denoting IPV4.
syscall
```

Create Socket

```
mov r12, rax         ← mov file descriptor generated by kernel for socket which is stored in rax to 'r12'
```

```
mov rax, 49          ← Sys call for 'bind' to bind the socket to a connection.
mov rdi, r12         ← file descriptor of socket from r12 to 'rdi' as 1st Parameter.
lea rsi, [rip+sockaddr]   ← Instruction stored at sockaddr, in 'rip' to be stored in rsi, entire data structure.
mov rdx, 16          ← size of the or length of socket address default '16'
syscall
```

Bind Socket

```
mov rdi, 0
mov rax, 60
syscall
```
} exit the socket after binding to look for new one.

for creating a seperate data structure to store the sockaddr_in data structure and call it in .section .text for execution.

```
.section .data
sockaddr:
    .word 2          # 2 byte default AF_INET
    .word 0x5000     # 2 byte Port no. stored in little endian hence written in opposite direction.
    .long 0          # 4 byte IP address here '0.0.0.0'
    .quad 0          # 8 byte or rest of byte padding with '0'.
```
Data structure for sock_addr_in in the bind function.

0x5000 → stored as 00 50 = (80) → Port 80

Little endian pattern ie. reverse storage.

→ Listen:

Now, our socket is formed and bind, so now it's waiting for connection, 'listen' syscall is used to listen for incoming connections and hence form connection. It transforms socket into passive one that awaits client's connection request. requires, → socket fd & backlog parameter, which sets maximum number of queued connections.

If Socket does not listen, server would'nt be able to recieve any connection attempts.

Now,

```
.intel_syntax noprefix
.global _start
.section .text
```

```
_start:
    mov rax, 41
    mov rdi, 2
    mov rsi, 1
    mov rdx, 0
    syscall
```
} create socket

```
mov r12, rax
```
→ store fd socket into 'r12'

```
    mov rax, 49
    mov rdi, r12
    lea rsi, [rip+addr_sock]
    mov rdx, 16
    syscall
```
} bind socket

syscall for listen.

```
    mov rax, 50
    mov rdi, r12
    mov rsi, 0
    syscall
```
} listen at socket with fd = value stored at 'r12'

listening; select socket with fd value = stored at r12.

backlog; logic set backlog to '0'.

```
    mov rdi, 0
    mov rax, 60
    syscall
```
} exit program or socket

```
.section .data
addr_sock:
    .word 2
    .word 0x5000
    .long 0
    .quad 0
```
} structure for bind.

→ Accept:

Now, our socket is in listening state so once it recieves a request for connection passively, we will actively accept the connection and form a active connection to do data transfer; for this we use 'accept' syscall.

When connection is established, it returns a new socket file descriptor (socket fd) dedicated to communication with that client and fills in a provided address structure. (such as struct sockaddr_in) with client's details.

Server $\xrightarrow{\text{listen}}$ Passive listen $\xrightarrow{\text{accept}}$ active communication

Now;

```
.intel_syntax noprefix
.global _start
 .section .text
 _start:.
  mov rax, 41
  mov rdi, 2
  mov rsi, 1
  mov rdx, 0
  syscall
```

} Create socket

mov r12, rax → store fd of socket in 'r12'.

```
  mov rax, 49
  mov rdi, r12
  lea rsi, [rip+sockaddr]
  mov rdx, 16
  syscall.
```

} bind Socket.

```
mov rax, 50
mov rdi, r12
mov rsi, 0
syscall
```
} listen on socket

```
mov rax, 43      ← Syscall for accept i.e. '43'.
mov rdi, r12     ← fd of socket is stored in rdi from 'r12'.
xor rsi, rsi     ← set rsi to NULL as it is acceptable
xor rdx, rdx     ← set rdx to NULL as it is acceptable.
Syscall
```
} accepts connection on socket.

```
mov rdi, 0
mov rax, 60
syscall
```
} exit.

```
.section .data
    sockaddr:
        .word 2
        .word 0x5000
        .long 0
        .quad 0
```
} struct for bind.

→ Static Response

Now, our server can establish successful connection with client, so now let's send fixed http response (HTTP/1.0 200 OK \n\n\r\n) to any client that connects.

for this we will use write syscall, which requires 'fd' a pointer to data buffer, no. of bytes to write.

Now;

```
.intel_syntax noprefix
.global _start
.section .text
_start:
    mov rax, 41
    mov rdi, 2
    mov rsi, 1
    mov rdx, 0
    syscall
```

Create Socket.

mov r12, rax → store fd of Socket into 'r12' = 3

```
    mov rax, 49
    mov rdi, r12
    lea rsi, [rip+sockaddr]
    mov rdx, 16
    syscall
```

bind Socket

```
    mov rax, 50
    mov rdi, r12
    mov rsi, 0
    syscall
```

listen Socket connection

```
    mov rax, 43
    mov rdi, r12
    xor rsi, rsi
    xor rdx, rdx
    syscall
```

accept Socket connection and new fd = '4'

mov r12, rax → update fd value to '4'

Syscall for read ← mov rax, 0

Set fd of new socket to read from fd = '4' ← mov rdi, r12

Set a buffer for storing the read value or request ← lea rsi, [rip+request_buf]

Size of buffer in the memory. ← mov rdx, 1024

Syscall

} read the value or request sent by client after connection establishment

Syscall for write. ← mov rax, 1

Set fd of new socket to write to. fd = '4' ← mov rdi, r12

Go to buffer set by read and store response to be sent ← lea rsi, [rip+response]

Size of message or response to be sent ← mov rdx, 19

Syscall.

} write the requested value back to client in the form of response.

Syscall for close ← mov rax, 3

Set fd to be closed. Here socket with fd='4' ← mov rdi, r12

Syscall

} close the connection of socket after response is made.

mov rdi, 0

mov rax, 60

Syscall.

} exit program.

Storing data structure in execution in .text ← .section .data

Sockaddr:

.word 2

.word 0x5000

.long 0

.quad 0

} struct for bind.

Write buffer to be written in response. ← response:

ascii conversion of given text string. ← .ascii "HTTP/1.0 200 OK\r\n\r\n"

Section for storing buffer memory for read. ← .section .bss

request_buf:

Set to 24 bytes to zero. ← .zero 1024

} read buffer.

→ Dynamic Response:

Now, let's make our server Dynamic, such that it accepts Dynamic http get requests and responds with the requested content. First we will use read syscall to recieve incoming http request from client socket.

Then we will examine request line - here URL path and determine what client is asking for. Then we will open requested file using open syscall and read contents of file and send contents to client using write syscall.

Now;

```
.intel-syntax noprefix
.global _start
.section .text
_start:
    mov rax,41
    mov rdi, 2
    mov rsi, 1
    mov rdx, 0
    syscall
```

Create Socket.

```
mov r12, rax
```

→ store fd of current socket. ie '3'

```
    mov rax, 49
    mov rdi, r12
    lea rsi, [ript sockaddr]
    mov rdx, 16
    syscall
```

bind socket.

```
    mov rax, 50
    mov rdi, r12
    mov rsi, 0
    syscall
```

listen for connection.

```
mov rax, 43
mov rdi, r12
xor rsi, rsi
xor rdx, rdx
syscall
```
} accept client connection on socket

mov r12, rax → store new fd of socket accepted. eg '4'

```
mov rax, 0
mov rdi, r12
lea rsi, [rip+ request_buf]
mov rdx, 1024
syscall
```
} read request sent by client and store it in buffer. request_buffer

make rsi is start of buffer i.e. request_buffer.

```
lea rsi, [rip+ request_buf]
```

Skip 1st 4 bytes of request stored in buffer i.e. "GET ". in order to only extract path from buffer.

```
add rsi, 4
```

Store modified buffer address to 'r13' to preserve it.

```
mov r13, rsi
```

```
parse_path_loop:
mov al, byte ptr [rsi]
```
make 'al' as current character in path.

```
cmp al, ' '
```
Compared 'al' with ' ' (blank) or end of path.

```
je .parse_path_done
```
If true jump to done parsing

```
cmp al, 0
```
else compare 'al' with '0'. '0' represents safety t end of buffer

```
je parse_path_done
```
if true jump to parsing done

```
inc rsi
```
else increment 'rsi' to move to next character position in path

```
jmp parse_path_loop
```
loop through all str again

} loop for parsing path stored in buffer from request

```
parse_path_done:
mov byte ptr [rsi], 0
```
} Done Parsing.

replace end of path with '\0' or null to end it.

Syscall for open ← mov rax, 2

name of path is path_start / take request URI processed from accepted fd stored in 'r13' ← mov rdi, r13

Permission of O_RDONLY ← mov rsi, 0

open read only.

mode. ← mov rdx, 0

Syscall

} Open the file from request recieved from fd of 'r13' accepted

store new fd for socket ← mov r13, rax

into 'r13' eg. '5'

Syscall for read ← mov rax, 0

fd selected at new ← mov rdi, r13

Set path of buffer ← lea rsi, [rip+file_buf]

at rsi at "file path".

exact storage of memory ← mov rdx, 4096

buffer holds.

Syscall.

} Read syscall for new updated fd after open file.

Store new fd with ← mov r14, rax

read stack of file path, to 'r14'

close syscall ← mov rax, 3

mov rdi, r13

Syscall

} close the 'fd' with socket having 'r13' fd - i.e. open file syscall.

mov rax, 1

mov rdi, r12

lea rsi, [rip+response]

mov rdx, 19

Syscall

} write syscall to write exactly '19' bytes into the buffer for response at fd 'r12'

mov rax, 1

mov rdi, r12

set 'rsi' to start of ← lea rsi, [rip+file_buf]

Buffer; file_buf

mov rdx, r14

Syscall.

} write syscall to write the read contents from request stored in 'r14' fd

```
mov rax, 3
mov rdi, r12
syscall
```
close the connection with fd 'r12' & response buffer.

```
mov rdi, 0
mov rax, 60
Syscall
```
exit program safely.

Data structure

```
.section .data
sockaddr:
    .word 2
    .word 0x5000
    .long 0
    .quad 0
```
structure for binding socket.

response data for reading and writing back.

```
response:
    .ascii "HTTP/1.0 200 OK\r\n\r\n"
```

buffers.

```
.section .bss
```

request buffer for reading request and storing it in buffer.

```
request_buf:
    .zero 1024
```

write buffer to write the file content into the buffer and to send it to the client.

```
file_buf:
    .zero 4096
```

→ Iterative GET Server:

Till now, our web server only accepts and serves only one get request and then terminates. So, Now we will modify it to serve multiple requests sequentially. We will put 'accept-read-write-close' sequence in a loop.

Now, we will keep the entire code same and only store socket fd in different register and accept fd in different for maintaing same socket to accept different connections. and then remove exit from end and add everything from accepting to writing into a loop.

So, Now keep all the code same and only perform following modifications;

```
.intel_syntax noprefix
.global _start
.section .text
_start:
    Socket initialization same code.
    mov rbx, rax  ---> store fd of socket in other register and not 'r12' as 'r12' gets rewritten further during accept.
    Now Perform binding
    and listening using 'rbx' as socket fd
    is 'rbx' instead of 'r12'
    Then;
    Server_loop:
        accept, Read, write for response
        and write for header all codes
        in this looped label.
        At the end replace exit with
        jmp Server_loop
```

To create a loop and write .data & .bss section as this.

→ Concurrent GET Server

Now, we will make our server deal with multiple users or clients simultaneously or concurrently using fork syscall to replicate the process and create a child instance which handles requests so that main server socket can listen for other clients.

Now, we have to close the web server's socket as soon as we get a request after creating a child process using fork to handle the get request of the particular client, for this, we will move all the read, write and close functionality to child process and terminate it by exit at the end of request and for parent as soon as child gets access of request, Parent will close that connection and look for new one to serve.

I will upload code in my repository named, "concurrent_get_server.s", look at it for reference.

→ Concurrent Post Server

Now, we will make our server deal with Post request concurrently. Post requests are more complex as they contain both header and message body. We will use fork for multiple connections and read to capture entire request. We will parse the URL to determine file, but instead of reading we will write to it using Post data. First we will determine length of incoming Post Data. Then we will write and save and then return a '200 OK' response to client to indicate Post request was succesful

Now; we will modify code accordingly and perform Post logic into it instead of 'GET' logic.
refer my repo saved file with name
"concurrent-post-servers".

→ Web Server:

Now, comes the fun part. Now our Server must support Dynamic flow of both GET & 'POST' requests.
Our Server must identify type of request by parsing first few characters or bytes of request; whether request is 'GET' or 'POST'
Try Doing This yourself or else look for it in my Projects repository.