COMPUTER SYSTEMS AND ORGANIZATION

Sockets

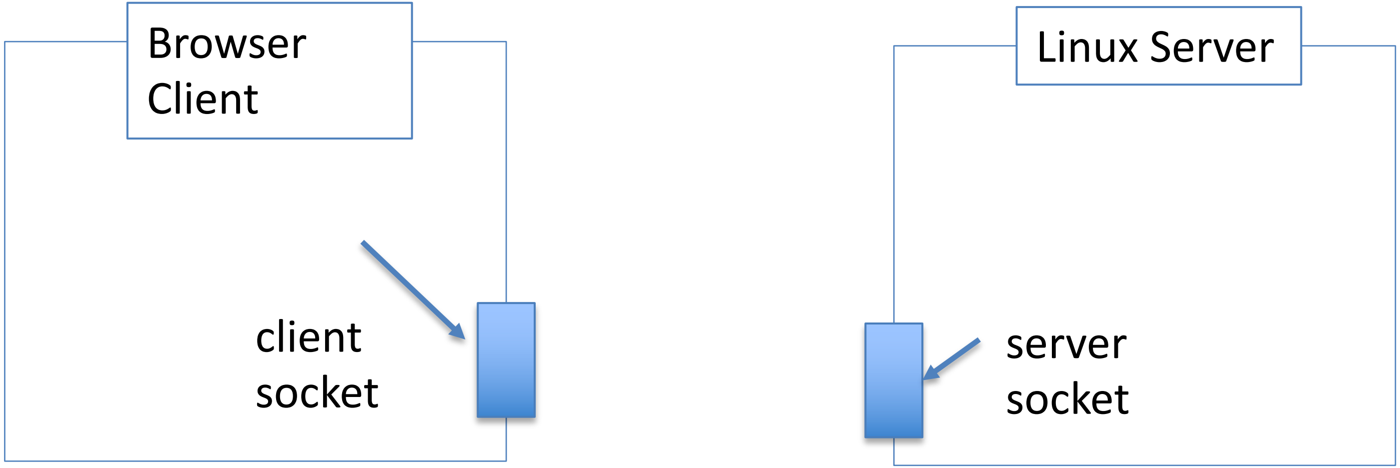
Daniel G. Graham Ph.D

UNIVERSITY of VIRGINIA | ENGINEERING

Contents

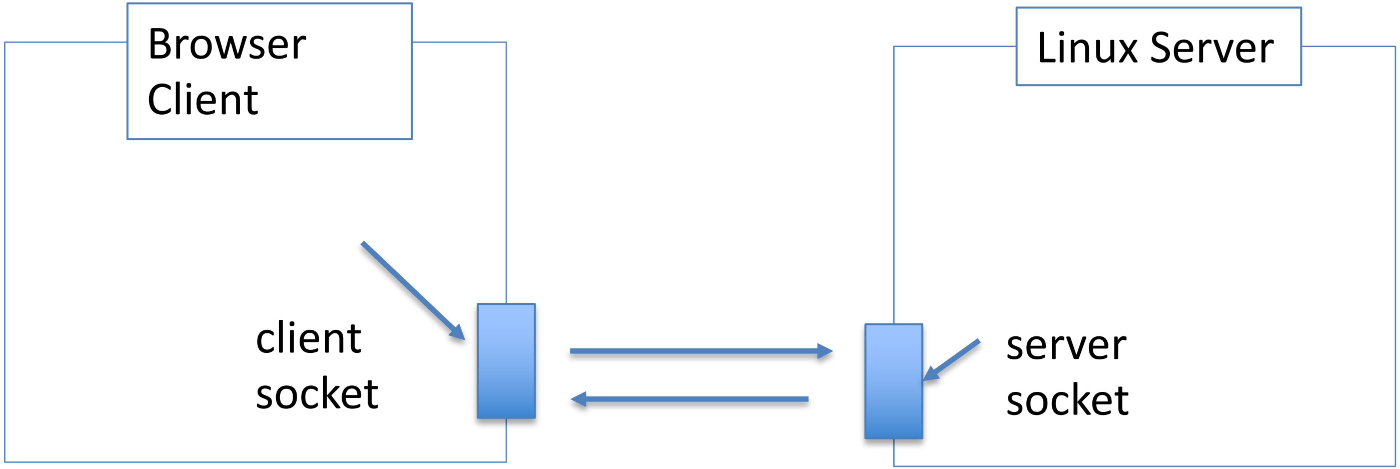
1. Client-server model
2. HTTP protocol basic
3. TCP Client
4. Client-server example demo
5. System Calls

CLIENT SERVER MODEL



Two types of sockets

CLIENT SERVER MODEL



Browser Client

client socket

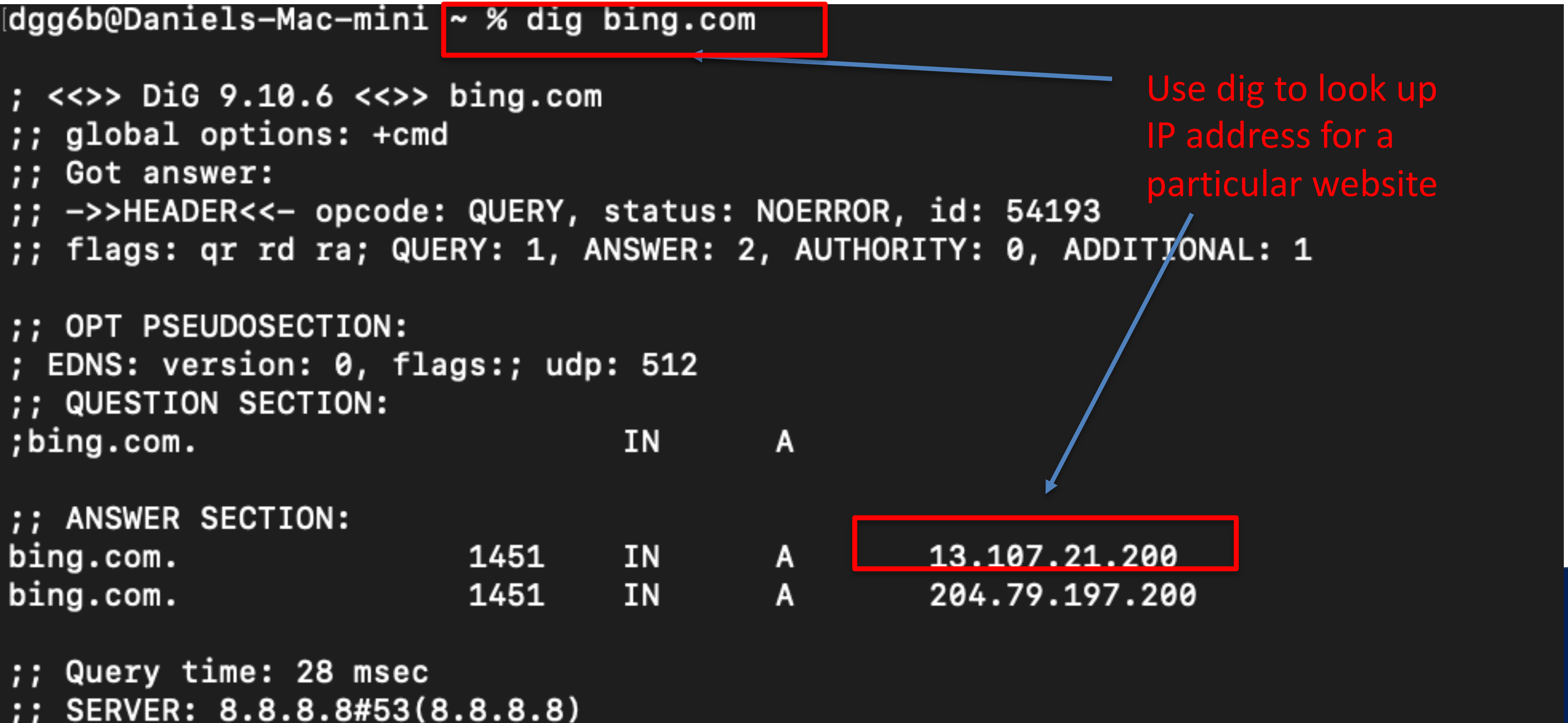
Linux Server

server socket

The network is abstracted From the programmer

Two types of sockets

DNS: FINDING THE IP FOR A DOMAIN



```
dgg6b@Daniels-Mac-mini ~ % dig bing.com

; <<>> DiG 9.10.6 <<>> bing.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 54193
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;bing.com.                      IN      A

;; ANSWER SECTION:
bing.com.               1451    IN      A       13.107.21.200
bing.com.               1451    IN      A       204.79.197.200

;; Query time: 28 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
```

Use dig to look up IP address for a particular website

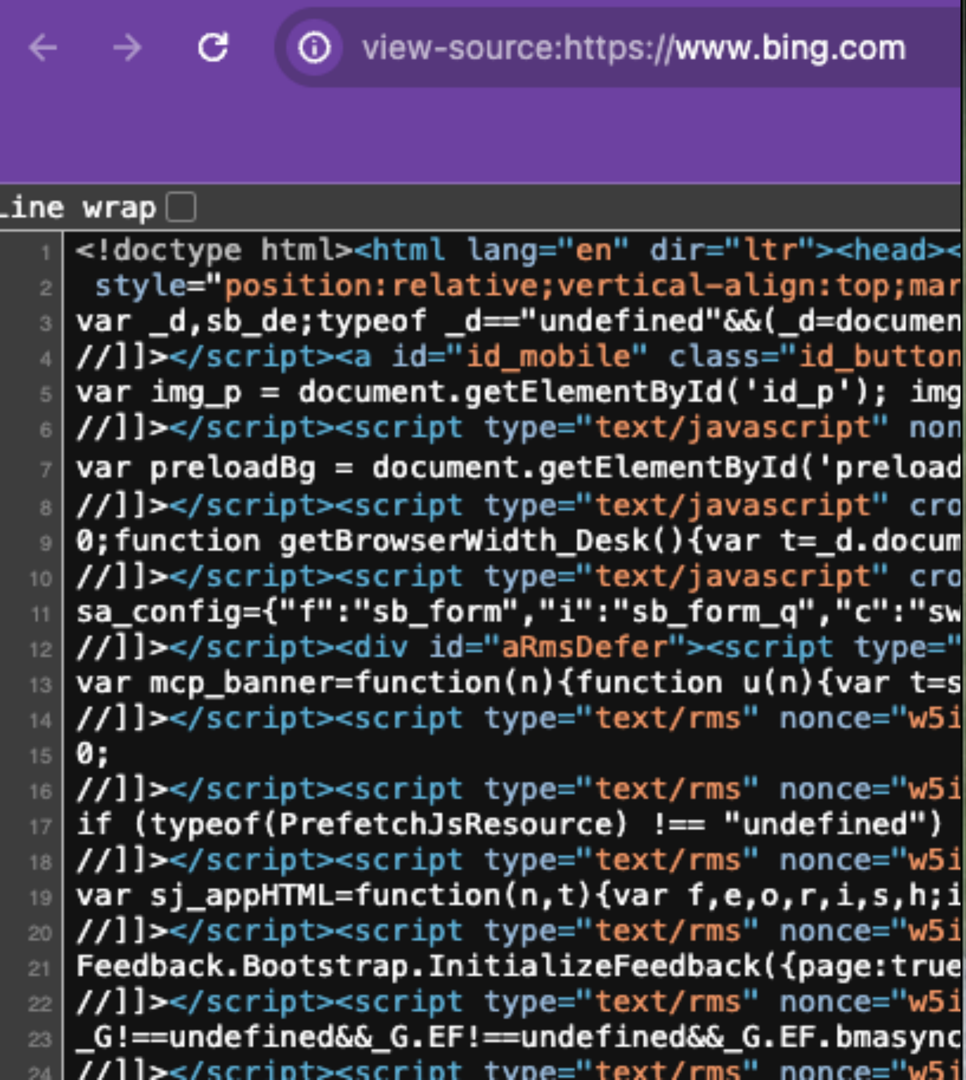
HTTP BASICS

REQUEST

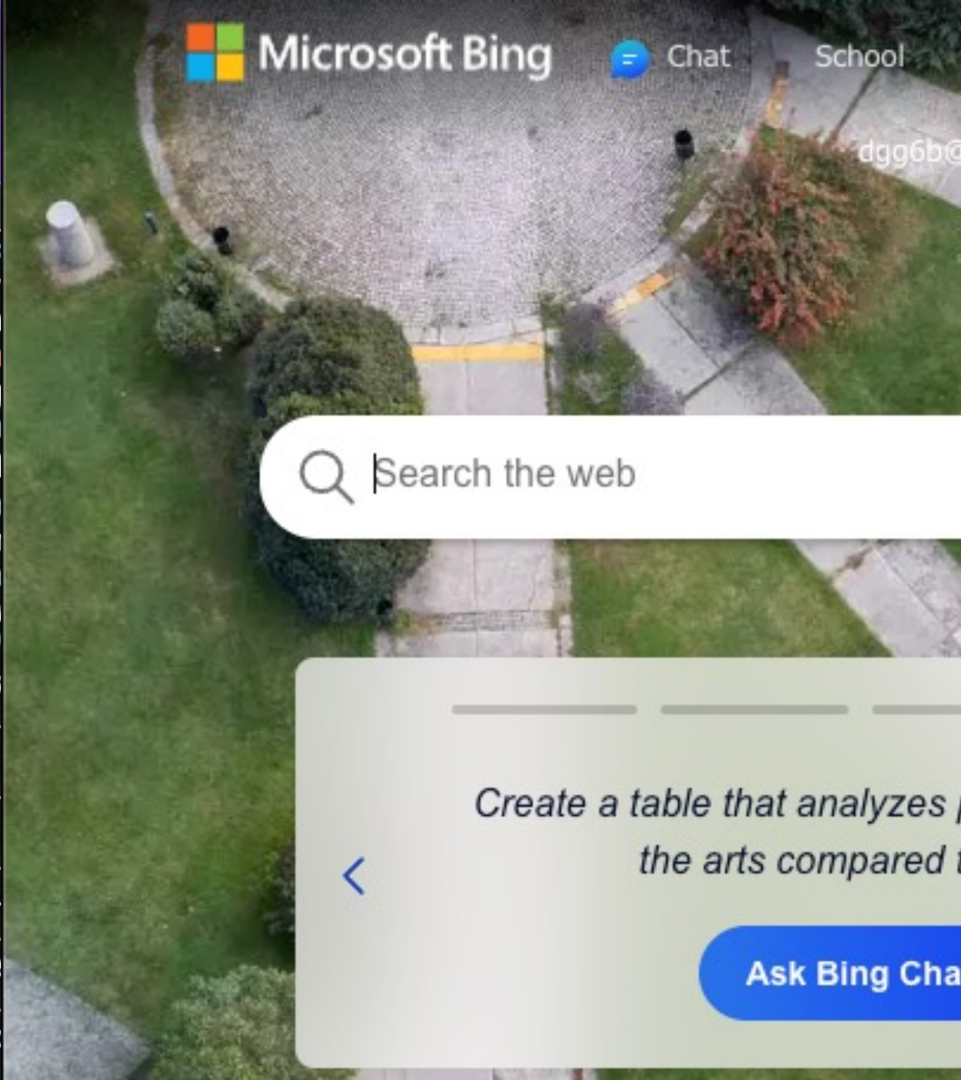
```
GET /index HTTP/1.1\r\n
Host: www.bing.com\r\n
\r\n
```

Response

```
HTTP/1.1 200 OK
--- Headers ---
--- Content ----
```



```
<!doctype html><html lang="en" dir="ltr"><head><
 style="position:relative;vertical-align:top;mar
var _d,sb_de;typeof _d=="undefined"&&(_d=documen
//]]></script><a id="id_mobile" class="id_button
var img_p = document.getElementById('id_p'); img
//]]></script><script type="text/javascript" non
var preloadBg = document.getElementById('preload
//]]></script><script type="text/javascript" cro
0;function getBrowserWidth_Desk(){var t=_d.docum
//]]></script><script type="text/javascript" cro
sa_config={"f":"sb_form","i":"sb_form_q","c":"sw
//]]></script><div id="aRmsDefer"><script type="
var mcp_banner=function(n){function u(n){var t=s
//]]></script><script type="text/rms" nonce="w5i
0;
//]]></script><script type="text/rms" nonce="w5i
if (typeof(PrefetchJsResource) !== "undefined")
//]]></script><script type="text/rms" nonce="w5i
var sj_appHTML=function(n,t){var f,e,o,r,i,s,h;i
//]]></script><script type="text/rms" nonce="w5i
Feedback.Bootstrap.InitializeFeedback({page:true
//]]></script><script type="text/rms" nonce="w5i
_G!==undefined&&_G.EF!==undefined&&_G.EF.bmasync
//]]></script><script type="text/rms" nonce="w5i
```



NOW LET’S WRITE A PROGRAM

Let’s write a c program that will send an HTTP request to the Bing servers and get the index page.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


#define PORT 80
#define BUFFER_SIZE 4096
#define SERVER_IP “13.107.21.200”

int main() {
    int sock;
    struct sockaddr_in server;
    char message[BUFFER_SIZE], response[BUFFER_SIZE];

    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);

    // Prepare the sockaddr_in structure
    server.sin_addr.s_addr = inet_addr("SERVER_IP");
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);

    // Connect to the server
    connect(sock, (struct sockaddr *)&server, sizeof(server));
```

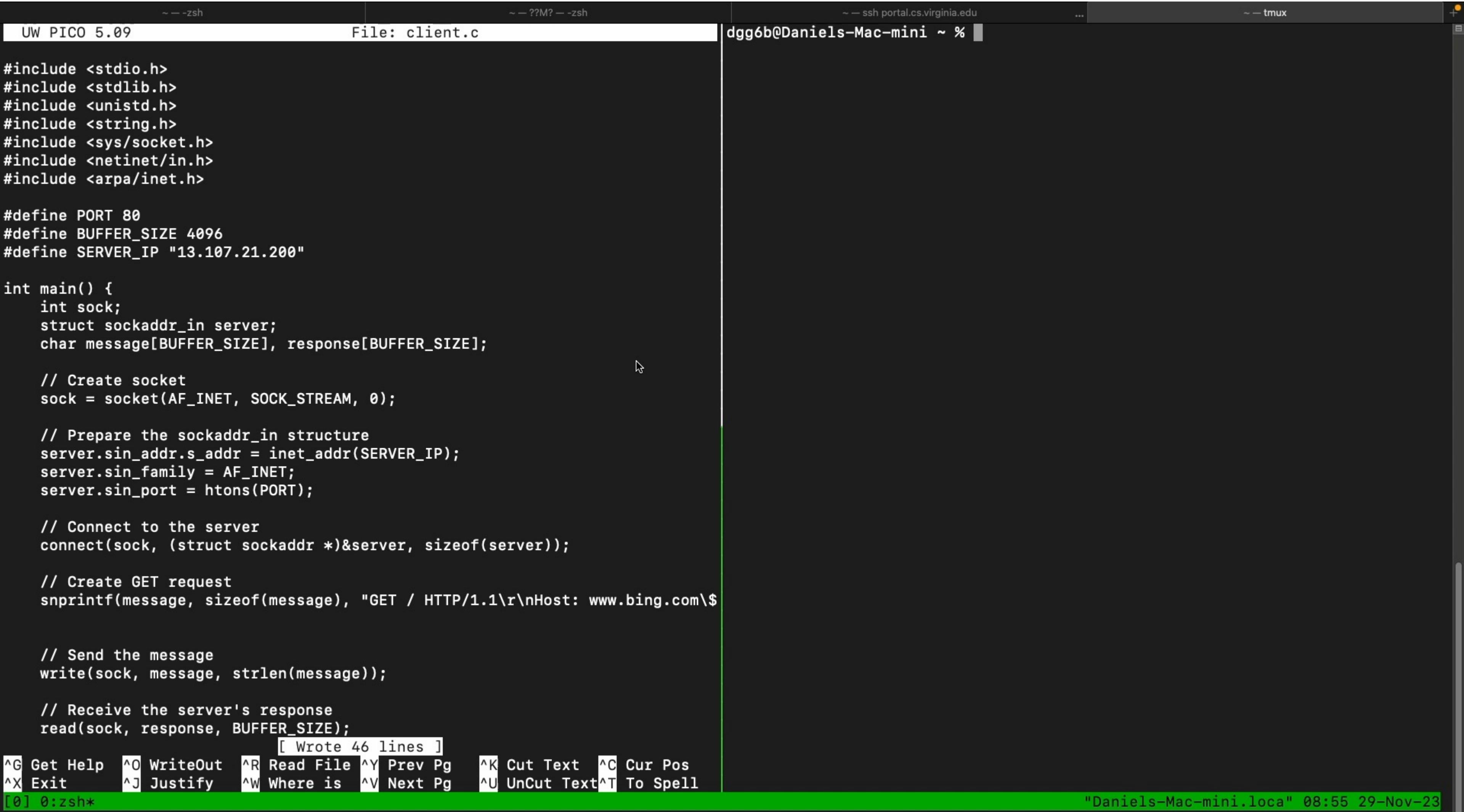
PART 1

Client vs Server:

Notice that we use connect instead of accept.

PART 2

```c
    // Create GET request
    snprintf(message, sizeof(message), "GET / HTTP/1.1\r\nHost: www.bing.com\r\n\r\n");


    // Send the message
    write(sock, message, strlen(message));

    // Receive the server's response
    read(sock, response, BUFFER_SIZE);

    printf("Server Response:\n%s", response);

    // Close the socket
    close(sock);

    return 0;
}
```

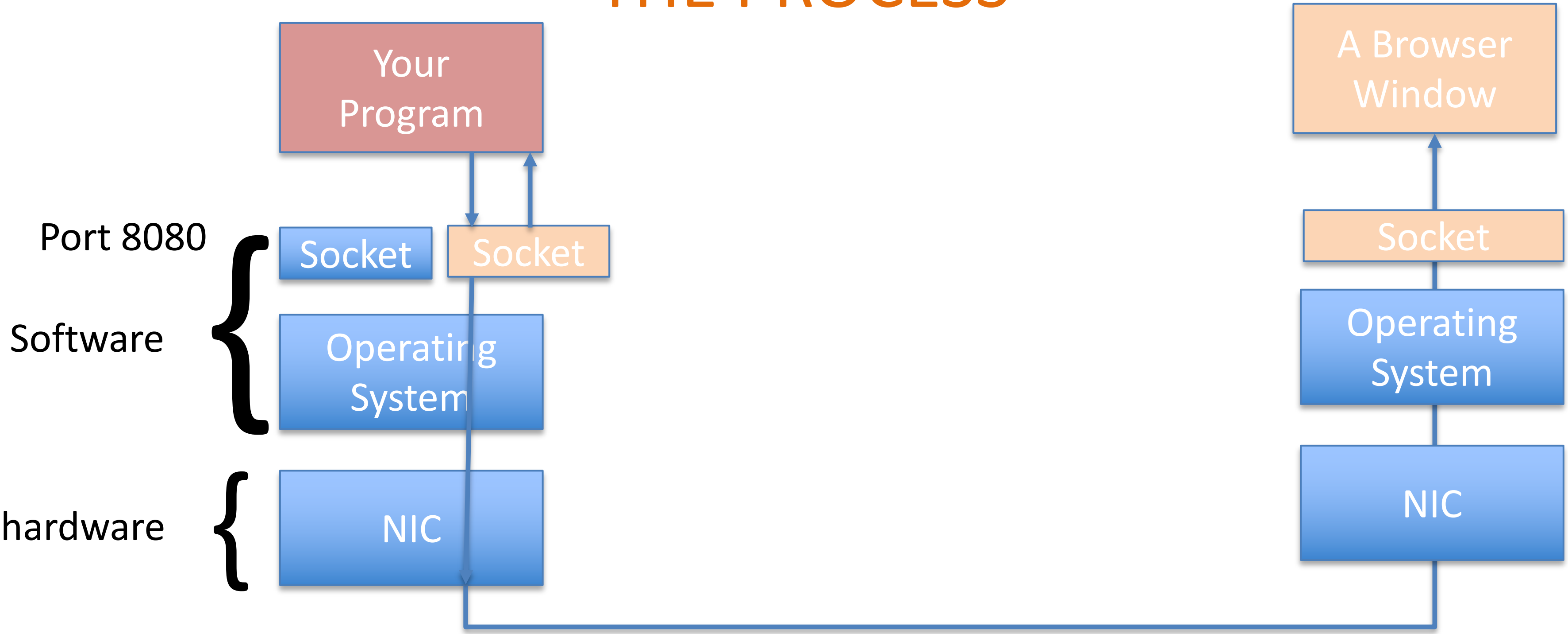


```
  UW PICO 5.09                  File: client.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 80
#define BUFFER_SIZE 4096
#define SERVER_IP "13.107.21.200"

int main() {
    int sock;
    struct sockaddr_in server;
    char message[BUFFER_SIZE], response[BUFFER_SIZE];

    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);

    // Prepare the sockaddr_in structure
    server.sin_addr.s_addr = inet_addr(SERVER_IP);
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);

    // Connect to the server
    connect(sock, (struct sockaddr *)&server, sizeof(server));

    // Create GET request
    snprintf(message, sizeof(message), "GET / HTTP/1.1\r\nHost: www.bing.com\$


    // Send the message
    write(sock, message, strlen(message));

    // Receive the server's response
    read(sock, response, BUFFER_SIZE);
                              [ Wrote 46 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Pg   ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is  ^V Next Pg   ^U UnCut Text^T To Spell
```

```
dgg6b@Daniels-Mac-mini ~ %
```

```
[0] 0:zsh*                                   "Daniels-Mac-mini.loca" 08:55 29-Nov-23
```

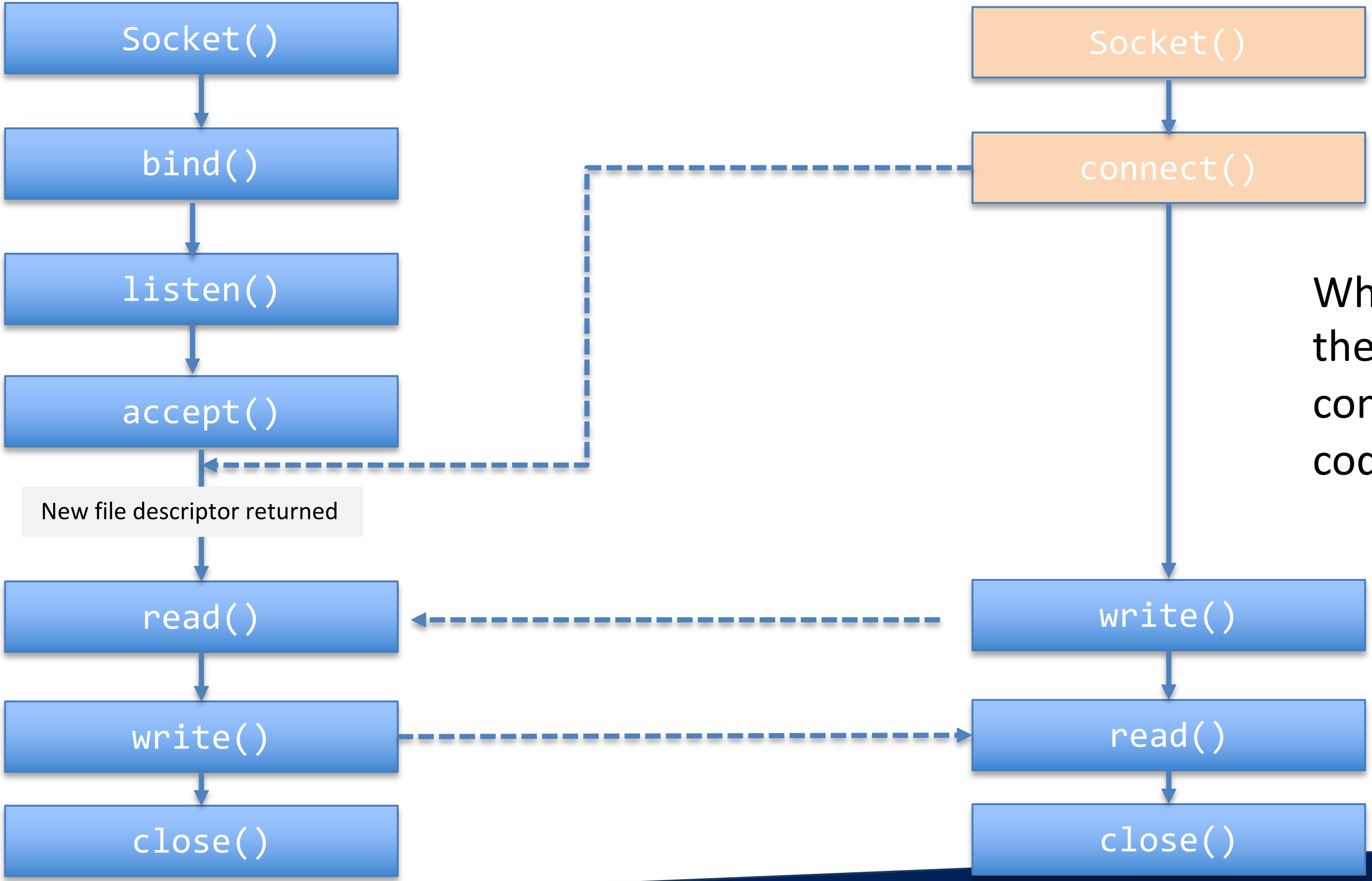
THE PROCESS


Your Program
Port 8080
Socket
Socket
Software
Operating System
hardware
NIC
A Browser Window
Socket
Operating System
NIC

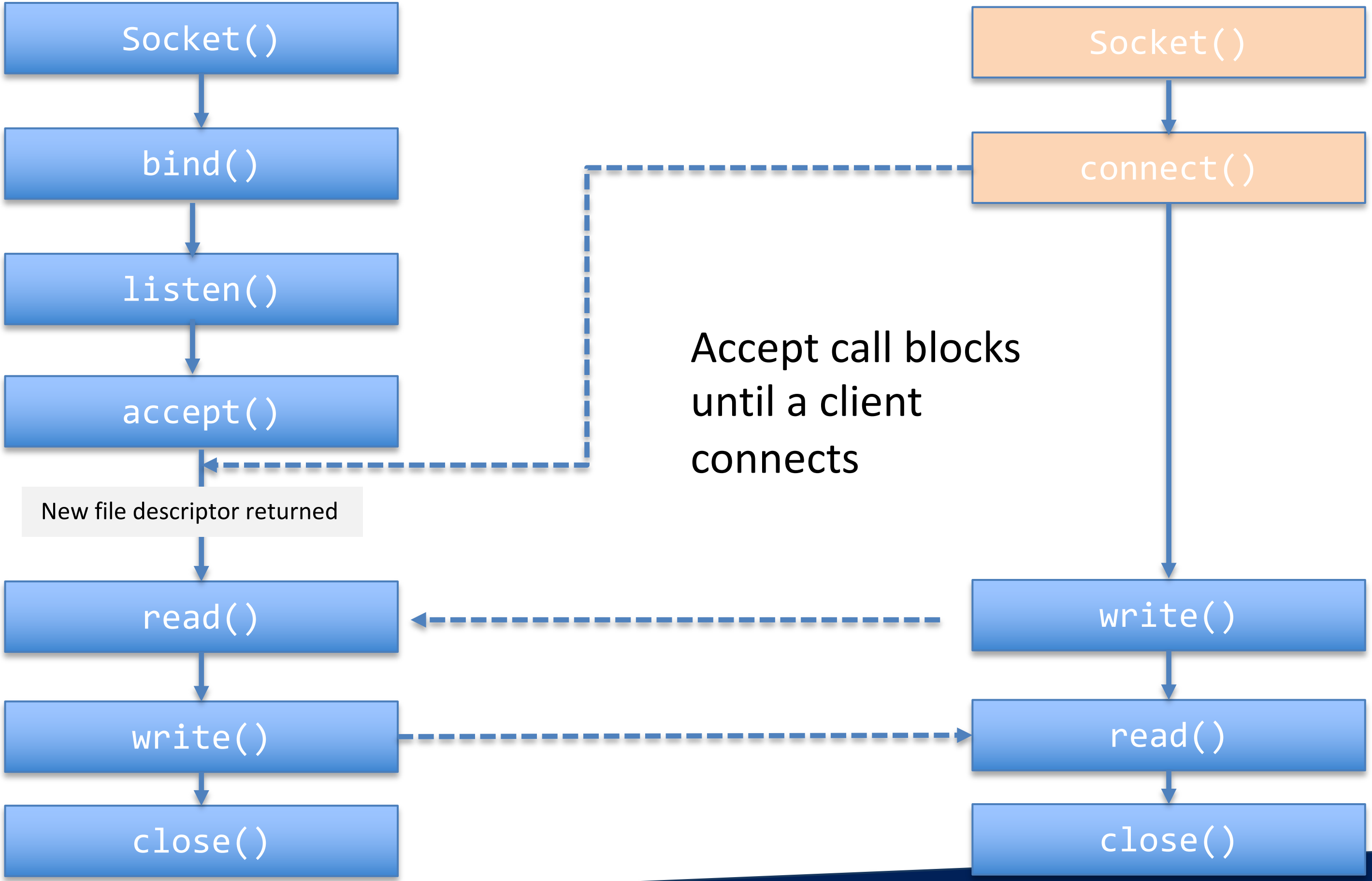
OUR SERVER

We have implemented both the client and server.

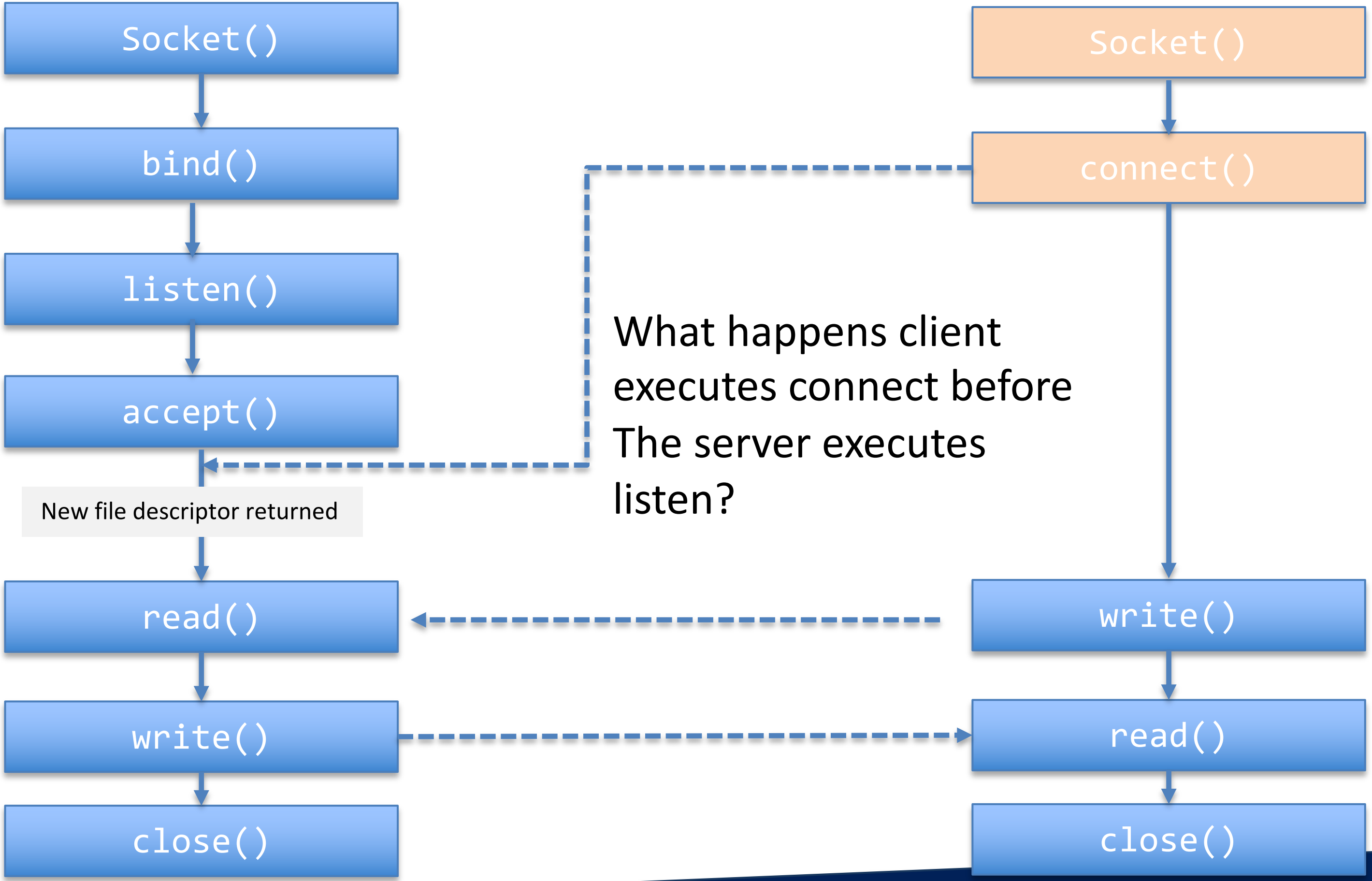
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>

#define PORT 8080

int main() {
    int server_fd, client_fd;
    struct sockaddr_in address;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 10);
    int addrlen = sizeof(address);

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen);
        write(new_socket, "HTTP/1.1 200 OK\n", 16);
        write(new_socket, "Content-Type: text/html\n\n", 25);
        write(new_socket, "<html><body><h1>Hello, World!</h1></body></html>", 44);
        close(new_socket);
    }
    close(server_fd);
    return 0;
}
```



What happens if the client get to connect a section of code before get t



Accept call blocks until a client connects



What happens client executes connect before The server executes listen?

WE HAVE BEEN USING FUNCTIONS LIKE WRITE HOW DOES THAT GET IMPLEMENTED IN ASSEMBLY?

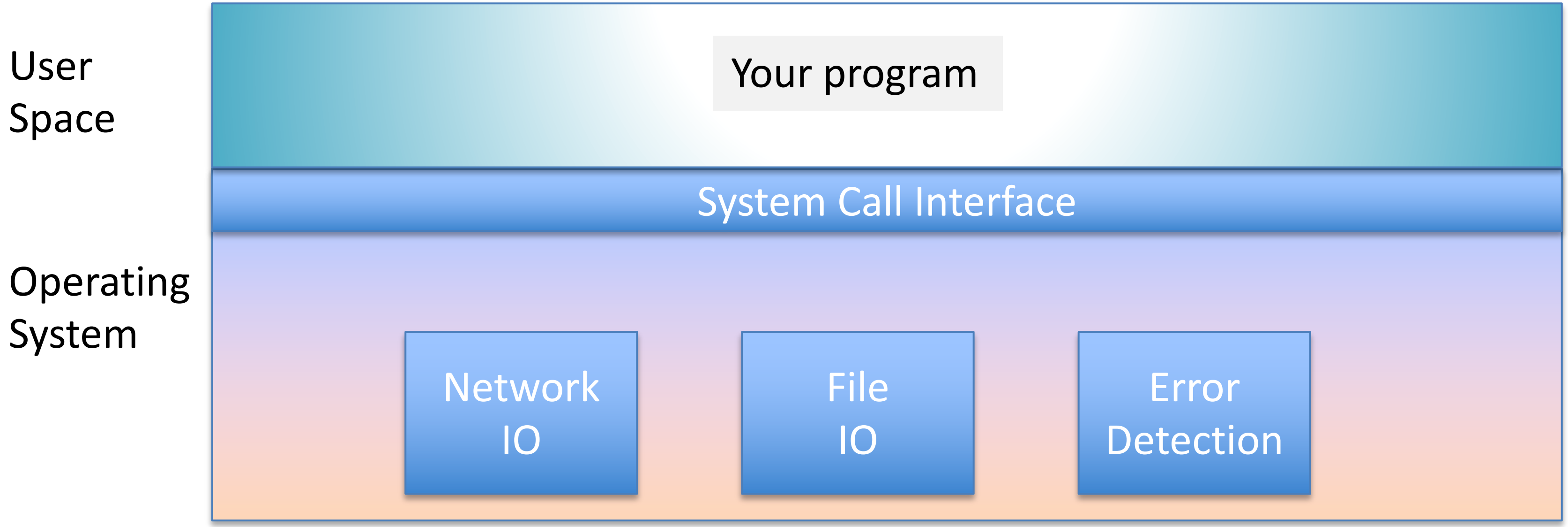
```
write(new_socket, "HTTP/1.1 200 OK\n", 16);
```

WHAT DOES THIS LOOK LIKE IN ASSEMBLY?

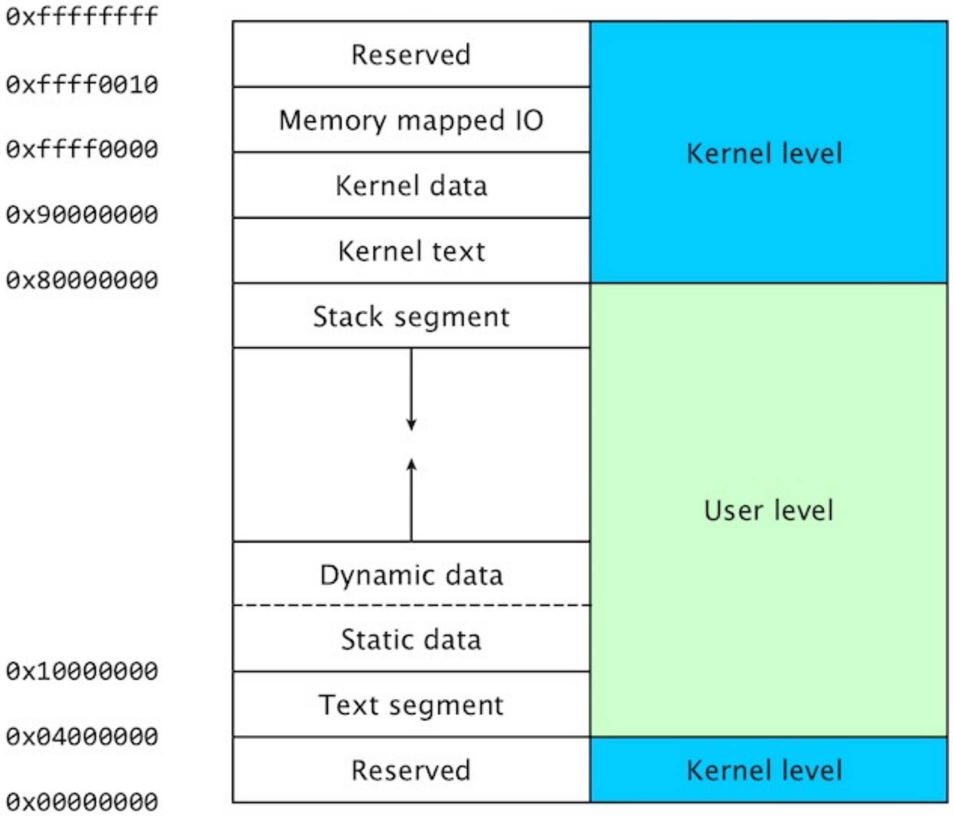
```
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    char *text = "CSO1";

    // Open a file for writing (create it if it doesn't exist)
    fd = open("output.txt", O_WRONLY | O_CREAT, 0644);

    // Write the string to the file
    write(fd, text, 4); // 4 is the number of bytes to write

    // Close the file
    close(fd);

    return 0;
}
```

Let’s look at this one. Sadly it not simple a call instruction to function located in fcntl

READING AND WRITING FILES AND THE NETWORK IS A PRIVILEGED OPERATION



USER SPACE VS KERNEL SPACE LINUX



Kernel layout for MIPS chips

https://www.it.uu.se/education/course/homepage/os/vt18/module-0/mips-and-mars/mips-memory-layout/

The layout of the arm chips can be found here. https://www.kernel.org/doc/html/v5.7/arm/memory.html

SYSTEM CALL CALLING CONVENTION

1. **Register Usage for Arguments**:
   1. %rax: System call number. Each system call has a unique number that you place in this register to tell the kernel which system call you're making.
   2. %rdi, %rsi, %rdx, %r10, %r8, %r9: Used for passing up to six arguments to system calls. %rdi is for the first argument, %rsi for the second, and so on. If a system call needs more than six arguments, a pointer to a block containing the arguments is passed as one of these registers.
2. **Making the System Call**:
   1. The syscall instruction is used to switch to kernel mode and invoke the system call. The kernel examines the value in %rax and understands which system call is being requested.
3. **Return Value**:
   1. After the system call, the return value is placed in %rax. This value typically indicates success or an error code.

THING ABOUT HOW YOU IMPLEMENT THE WRITE SYSTEM CALL TO STDOUT

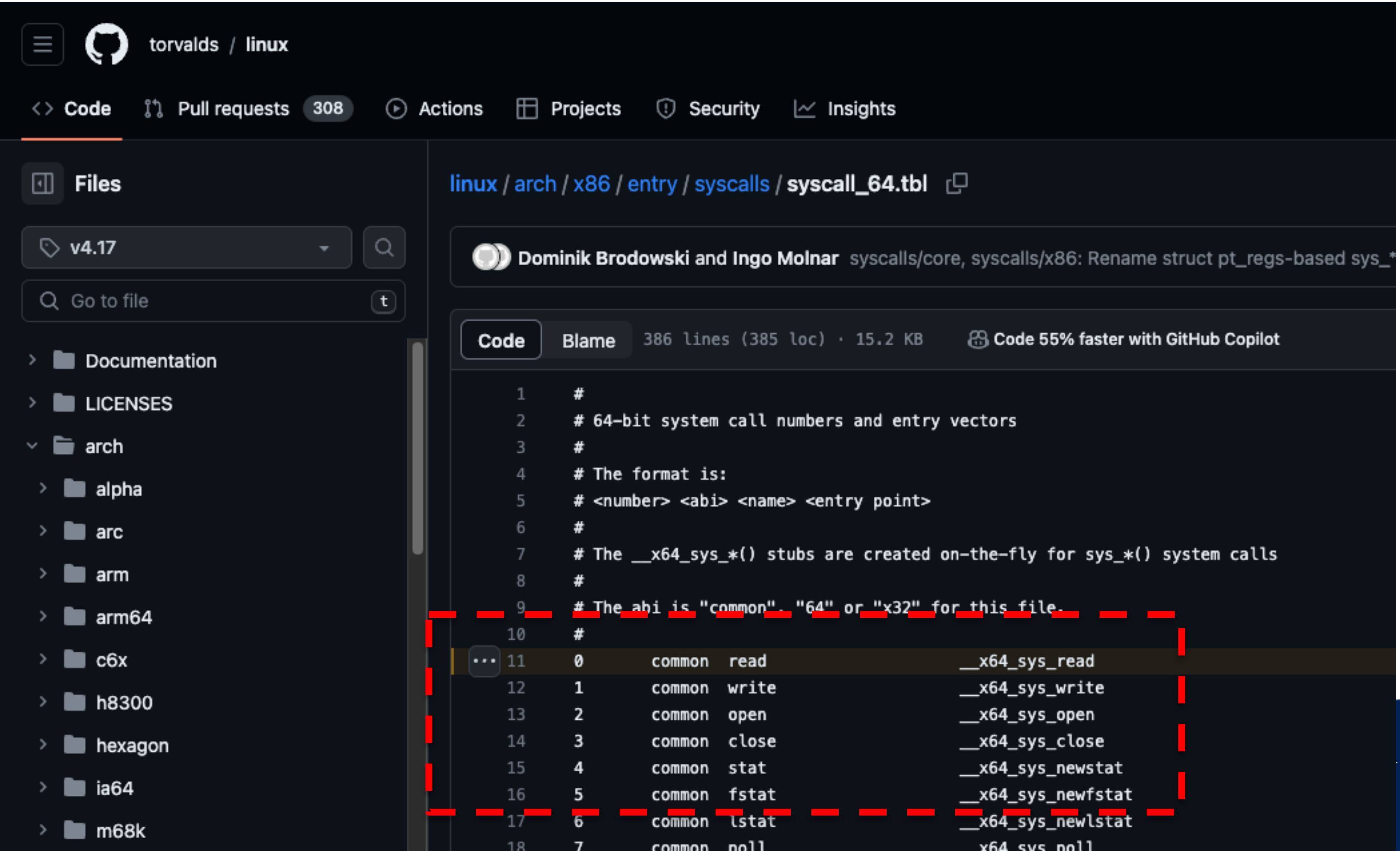
```
write(1, message, message_length)
```

1. **Register Usage for Arguments**:
   1. %rax: System call number. Each system call has a unique number that you place in this register to tell the kernel which system call you're making.
   2. %rdi, %rsi, %rdx, %r10, %r8, %r9: Used for passing up to six arguments to system calls. %rdi is for the first argument, %rsi for the second, and so on. If a system call needs more than six arguments, a pointer to a block containing the arguments is passed as one of these registers.
2. **Making the System Call**:
   1. The syscall instruction is used to switch to kernel mode and invoke the system call. The kernel examines the value in %rax and understands which system call is being requested.
3. **Return Value**:
   1. After the system call, the return value is placed in %rax. This value typically indicates success or an error code.

SYSTEM CALL CALLING CONVENTION

```
.global _start
.text
_start:
    # write(1, message, 18)
    mov     $1, %rax                 ; syscall number for write (1)
    mov     $1, %rdi                 ; file descriptor 1 (stdout)
    lea     message(%rip), %rsi      ; load the address of the message
    mov     $18, %rdx                ; message length (18 bytes)
    syscall                          ; perform the system call


.section .rodata                     ; Read-only data section
message:                             ; Label for the message
    .ascii  "Computer Systems 1"     ;
```

WHERE CAN I FIND THE SYSTEM CALL NUMBERS



Dominik Brodowski and Ingo Molnar syscalls/core, syscalls/x86: Rename struct pt_regs-based sys_*

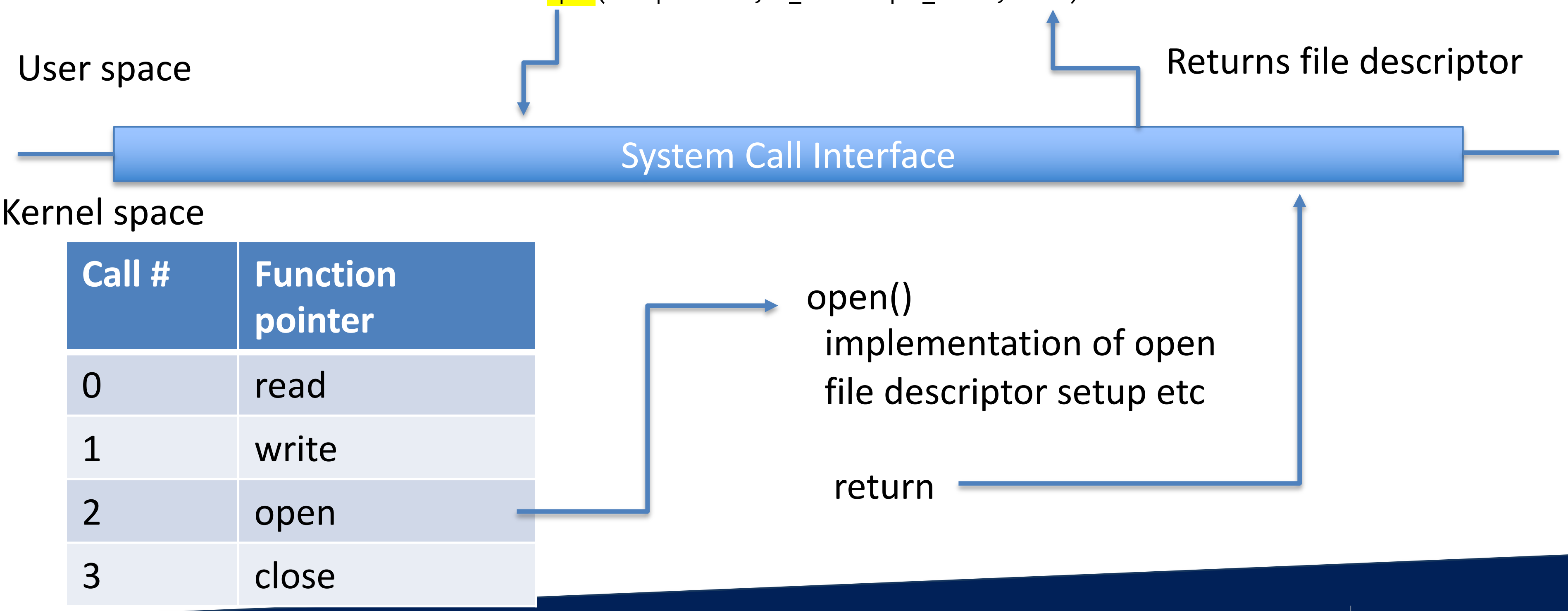
```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    __x64_sys_read
1       common  write                   __x64_sys_write
2       common  open                    __x64_sys_open
3       common  close                   __x64_sys_close
4       common  stat                    __x64_sys_newstat
5       common  fstat                   __x64_sys_newfstat
6       common  lstat                   __x64_sys_newlstat
7       common  poll                    __x64_sys_poll
```

Linux github repo.

https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_64.tbl

SYSTEM CALLS

```
int main() {
    int fd;
    char *text = "CSO1";

    // Open a file for writing (create it if it doesn't exist)
    fd = open("output.txt", O_WRONLY | O_CREAT, 0644)
```



User space

Returns file descriptor

System Call Interface

Kernel space

Call #	Function pointer
0	read
1	write
2	open
3	close

open()
  implementation of open
  file descriptor setup etc

return

WHAT DOES THE FOLLOWING ASSEMBLY DO?

```
.global _start
.text
_start:
    # What does this snipet of assemble do?
    mov     $3, %rax                ;
    mov     $1, %rdi                ;
    syscall                         ;
```

Call #	Function pointer
0	read
1	write
2	open
3	close

A. Write Perror
B. Write stdout
C. Open stdout
D. Open Perror
E. Read from Perror
F. Close Perror
G. Read stdout
H. Close stdout
I. Read stdin
J. Close std in