Ping&watchdog - programming in C

Authors:

Maor Saadon 318532421

Dovi Amiram 305677494

Contents

1 System Characterization

1.1 System Overview

1.1.1 About the System

Part A:

The ping command is used to check the connection between 2 machines. In part A, we will implement the “ping” command.

we will write a program called “ping.c” which will get an argument indicating which host to ping.

The Internet Control Message Protocol (ICMP) is a supporting protocol in the Internet protocol suite. It is used by network devices, including routers, to send error messages and operational information indicating success or failure when communicating with another IP address, for example, an error is indicated when a requested service is not available or that a host or router could not be reached. ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, nor is it regularly employed by end-user network applications (with the exception of some diagnostic tools like ping and traceroute).

ICMP ECHO REQUEST AND ICMP-ECHOREPLY - The ICMP echo request and the ICMP echo reply messages are commonly known as ping messages. Ping is a troubleshooting tool used by system administrators to manually test for connectivity between network devices, and also to test for network delay and packet loss.

The program will send an ICMP ECHO REQUEST to the host, and when receiving ICMP-ECHOREPLY, the program will send the next ICMP ECHO REQUEST (no need to stop).

Part B:

Watchdog is a timer to detect and recover your computer dis-functions or hardware fails. It's a chip whose sole purpose is to receive a signal every millisecond from the CPU.

It will reboot the system if it hasn't received any signal for 10 seconds (mostly when hardware fails).

We will modify the ping program and write a watchdog that will hold a timer (TCP connection on port 3000) to ensure that if we don't receive an ICMP-ECHO-REPLY after sending an ICMP-REQUEST for 10 seconds, it will exit.

We will modify the ping.c program so that it will execute the watchdog.c program as well using fork + exec.

Every time better_ping.c sends a packet, we will update watchdog.c timer.

1.1.2 How to Install and Run the Program

To test the system for yourself, you would need a Linux based operating system.

Instructions:

1. Download the following files:
    a. ping.c
    b. better_ping.c
    c. watchdog
    d. Makefile
2. Put all of the above files in a single directory.
3. Open said directory in your Linux terminal.
4. Run the following commands:
    a. sudo apt install build-essential
    b. Make all
    c. sudo ./parta <IP> - for part A
    d. sudo ./partb <IP> - for part B
5. When you want to close Part A you need to press on Ctrl+c

1.2 System Functionality

1.2.1 Code Description

Part a:

ping.c:

1. The program defines the ICMP header length:

```
#define ICMP_HDRLEN 8// ICMP header len for echo req
```

2. Next, the program declares 3 functions implemented below the main code (see detailed explanation below):

```
//function declaration
unsigned short calculate_checksum(unsigned short *paddress, int len);
int validateNumber(char *str);
int validateIp(char *ip);
```

The calculate_checksum() function calculates the checksum of an ICMP header. The validateNumber() function checks if a string is a valid number. The validateIp() function checks if a string is a valid IP address.

3.

```
int main(int argc, char *argv[]){
    //check the IP
    if (argc != 2) {
        perror("you need to put IP!");
        exit(-1);
    }
    char ptr_IP[15];
    strcpy(ptr_IP , argv[1]);
    int flage = validateIp(ptr_IP);
    if (flage == 0)
    {
        printf("Ip isn't valid!\n");
        exit(-1);
    }
```

The main function takes two arguments: the number of command-line arguments (argc) and an array of strings containing the arguments (argv). The program first checks if the user

provided exactly one command-line argument (the IP address). If not, it prints an error message and exits. It then copies the IP address from argv[1] to the string ptr_IP, and checks if the IP address is valid using the validateIp function. If the IP address is not valid, it prints an error message and exits.

4. Creating a raw socket:

```
//open socket
int sock = -1;
if ((sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1) {
    fprintf(stderr, "socket() failed with error: %d", errno);
    fprintf(stderr, "To create a raw socket, the process needs to be run by Admin/root user.\n\n");
    return -1;
}
```

The socket() function creates a new socket and returns a file descriptor for it (for a more detailed explanation – see below).

5. The program then sets up the destination address for the socket:

```
    struct sockaddr_in dest_in;
    memset(&dest_in, 0, sizeof(struct sockaddr_in));
    dest_in.sin_family = AF_INET;
    dest_in.sin_addr.s_addr = inet_addr(argv[1]);// The port is irrelant for Networking and therefore was
 zeroed.
```

It does this by creating a sockaddr_in structure and setting the address family to AF_INET and the IP address to the provided IP address.

6. Declaring an ICMP header structure and creating a data packet containing a message::

```
struct icmp icmphdr; // ICMP-header
char data[IP_MAXPACKET] = "This is the ping.\n";
int datalen = strlen(data) + 1;
```

The IP_MAXPACKET macro is the maximum size of an IP packet. The data packet is a string containing a message, and the datalen variable is the length of the data packet.

7. Printing a message indicating that it is pinging the target IP address:

```
printf("PING %s (%s): %d data bytes\n", argv[1], argv[1], datalen);
```

8. Declaring a timeval structure for measuring the round-trip time of each ping:

```
//for calculate the time
struct timeval start, end;
gettimeofday(&start, 0);
int  icmp_seq_counter = 0;
```

The gettimeofday function gets the current time and stores it in the start structure.

9. The program then enters a loop where it sends a ping, waits for a reply, and prints a message based on whether a reply was received or not. It also calculates the round-trip time for each ping and keeps track of the number of successful and failed pings:

```
    while (1) {

        //ICMP header
        //___________
        icmphdr.icmp_type = ICMP_ECHO;// Message Type (8 bits): ICMP_ECHO_REQUEST
        icmphdr.icmp_code = 0;// Message Code (8 bits): echo request
        // Identifier (16 bits): some number to trace the response.
        // It will be copied to the response packet and used to map response to the request sent earlier.
        // Thus, it serves as a Transaction-ID when we need to make "ping"
        icmphdr.icmp_id = 18;
        icmphdr.icmp_seq = 0;// Sequence Number (16 bits): starts at 0
        icmphdr.icmp_cksum = 0;// ICMP header checksum (16 bits): set to 0 not to include into checksum
calculation
```

The ICMP header is initialized with the following values:

a. icmp_type: set to ICMP_ECHO to indicate that this is an echo request packet
b. icmp_code: set to 0
c. icmp_id: set to 18
d. icmp_seq: set to 0
e. icmp_cksum: set to 0

10. Combining the ICMP header and the data packet into a single packet:

```
char packet[IP_MAXPACKET];// Combine the packet
memcpy((packet), &icmphdr, ICMP_HDRLEN); // Next, ICMP header
memcpy(packet + ICMP_HDRLEN, data, datalen);// After ICMP header, add the ICMP data.
```

11. Calculating the checksum of the ICMP header and stores it in the header:

```
icmphdr.icmp_cksum = calculate_checksum((unsigned short *) (packet),
                                        ICMP_HDRLEN + datalen);// Calculate the ICMP header checksum
memcpy((packet), &icmphdr, ICMP_HDRLEN);
```

12. Waiting for 1 second before sending the packet:

```
sleep(1);
```

13. Sending the packet using the sendto() function (see detailed explaination below):

```
// Send the packet using sendto() for sending datagrams.
int bytes_sent = sendto(sock, packet, ICMP_HDRLEN + datalen, 0, (struct sockaddr *) &dest_in, sizeof(dest_in));
if (bytes_sent == -1) {
    fprintf(stderr, "sendto() failed with error: %d", errno);
    return -1;
}
```

14. Clearing 'packet' char array and calling the recvfrom function is to receive a packet from the host.

```
// Get the ping response
bzero(packet, IP_MAXPACKET);
socklen_t len = sizeof(dest_in);
ssize_t bytes_received = -1;
while ((bytes_received = recvfrom(sock, packet, sizeof(packet), 0, (struct sockaddr *) &dest_in, &len))) {
    if (bytes_received > 0) {
        // Check the IP header
        struct iphdr *iphdr = (struct iphdr *) packet;
        struct icmphdr *icmphdr = (struct icmphdr *) (packet + (iphdr->ihl * 4));
        break;
    }
}
```

The recvfrom function returns the number of bytes received, or -1 if an error occurs. The loop continues until a response is received (i.e., bytes_received is greater than 0).

15. Once a response is received, the IP header and ICMP header are extracted from the packet:

```
if (bytes_received > 0) {
    // Check the IP header
    struct iphdr *iphdr = (struct iphdr *) packet;
    struct icmphdr *icmphdr = (struct icmphdr *) (packet + (iphdr->ihl * 4));
    break;
}
```

The IP header is at the beginning of the packet, and the ICMP header follows the IP header. The length of the IP header is specified in the ihl field of the iphdr structure.

16. Finally, the round-trip time is calculated and printed:

```
    gettimeofday(&end, 0);

    //calculate the time
    float milliseconds = (end.tv_sec - start.tv_sec) * 1000.0f + (end.tv_usec - start.tv_usec) / 1000.0f;
    printf("%ld bytes from %s: icmp_seq=%d ttl=10 time=%f ms)\n", bytes_received, argv[1], icmp_seq_counter++,
           milliseconds);
}
```

17. Eventually close the socket and return 0 to the main function.

```
// Close the raw socket descriptor.
close(sock);

return 0;
```

Part b:

better_ping:

Get in the main function Ipv4 address as an argument indicating which host to ping.
It sends an ICMP ECHO REQUEST to the host and receives the response by ICMP-ECHOREPLY.
The sending happens repeatedly until the program terminated, or until the watchdog close the program.

1. **Check the IP address:**
   If the IP is a valid IPv4 address we continue. If not, the program terminated with an error massage.

```
if (argc != 2) {
    perror("you need to put IP!");
    exit(-1);
}
char ptr_IP[15];
strcpy(ptr_IP , argv[1]);
int flage = validateIp(ptr_IP);
if (flage == 0)
{
    printf("Ip isn't valid!\n");
    exit(-1);
}
```

2. **run 2 programs using fork + exec:**
   To run both better_ping and watchdog in parallel we used fork() that creates a new process by duplicating the calling process. The new process can be called the child process end the calling process can be called the parent process. The child and the parent processes are running in different space in the memory.
   At the time of fork() both memory spaces have the same content. Memory writes, file mappings performed by one of the processes do not affect the other.
   In our case, the parent process is the better_ping, and the child process is the watchdog.

```
char *args[2];
// compiled watchdog.c by makefile
args[0] = "./watchdog";
args[1] = argv[1];
int status;
int pid = fork();
if (pid == 0)
{
    execvp(args[0], args);
}
```

3. **Creating TCP connection between the better_ping and watchdog.**
    a. we create a new TCP socket by using the socket() method (like we created in the last assignment Ex3).
    b. Fill the serverAdrees (watchdog in our case) with the right value.
    .sin_family - is an address family that is used to designate the type of addresses that the socket can communicate with. In this case, Internet Protocol v4 addresses.
    .sin_port - refers to a 16-bit port number on which the watchdog will listen to the connection requests by the better_ping. (htons() convert the port to network endian)
    inet_pton() method: convert IPv4 addresses from text which written in SERVER_IP_ADDRESS to binary form and save the convertion in serverAddress.sin.addr.
    c. Make connection with watchdog by using connect() method (like we created in the last assignment Ex3).

```c
//open a socket for watchdog
int watchdogSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (watchdogSocket == -1) {
    printf("Could not create socket : %d\n", errno);
    return -1;
}
struct sockaddr_in serverAddress;
memset(&serverAddress, 0, sizeof(serverAddress));
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(WATCHDOG_PORT);
int rval = inet_pton(AF_INET, (const char *) SERVER_IP_ADDRESS, &serverAddress.sin_addr);
if (rval <= 0) {
    printf("inet_pton() failed");
    return -1;
}

sleep(3);

// Make a connection with watchdog
int connectResult = connect(watchdogSocket, (struct sockaddr *) &serverAddress, sizeof(serverAddress));
if (connectResult == -1) {
    printf("connect() failed with error code : %d\n", errno);
    close(watchdogSocket);
    return -1;
}
```

4. **Creating a RAW socket connection between the better.ping:**

a. In order to send ping massages to the IP address we need a row socket, the raw socket is a type of socket that allows access to the underlying transport provider. This topic focuses only on raw sockets and the IPv4 and IPv6 protocols. This is because most other protocols (apart from ATM) do not support raw sockets.
b. .sin_family - is an address family that is used to designate the type of addresses that the socket can communicate with. In this case, Internet Protocol v4 addresses.
c. .sin_adder.s_adder – inet_addr() converts IPv4 (the user input) address in dot-decimal notation to a 32-bit binary number in network byte order.

```
//open socket for ping
int sock = -1;
if ((sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1) {
    fprintf(stderr, "socket() failed with error: %d", errno);
    fprintf(stderr, "To create a raw socket, the process needs to be run by Admin/root user.\n\n");
    return -1;
}
struct sockaddr_in dest_in;
memset(&dest_in, 0, sizeof(struct sockaddr_in));
dest_in.sin_family = AF_INET;
dest_in.sin_addr.s_addr = inet_addr(argv[1]);// The port is irrelant for Networking and therefore was zeroed.

// ICMP-header
struct icmp icmphdr;
char data[IP_MAXPACKET] = "This is the ping.\n";
int datalen = strlen(data) + 1;
```

5. **like we already explained in ping.c:**

```
icmphdr.icmp_type = ICMP_ECHO;// Message Type (8 bits): ICMP_ECHO_REQUEST
icmphdr.icmp_code = 0;// Message Code (8 bits): echo request
icmphdr.icmp_id = 18;
icmphdr.icmp_seq = 0;// Sequence Number (16 bits): starts at 0
icmphdr.icmp_cksum = 0;// ICMP header checksum (16 bits): set to 0 not to include into checksum calculation
char packet[IP_MAXPACKET];// Combine the packet
memcpy((packet), &icmphdr, ICMP_HDRLEN); // Next, ICMP header
memcpy(packet + ICMP_HDRLEN, data, datalen);// After ICMP header, add the ICMP data.
icmphdr.icmp_cksum = calculate_checksum((unsigned short *) (packet),
                                        ICMP_HDRLEN + datalen);// Calculate the ICMP header checksum
memcpy((packet), &icmphdr, ICMP_HDRLEN);
```

6. send an ack message to watchdog, in order to notify him to start calculating the time. So, he will be able to know if he should terminate the whole program.

```
//if NEW_PING don't receive a stop signal -> send to watch dog that NEW_PING ready to start sending ping
int ready = 1;
int s = send(watchdogSocket, &ready, sizeof(int), 0);

gettimeofday(&start, 0);
```

7. we call the sleep function in ordr to test if the watchdog terminte the program when he should.

```
gettimeofday(&start, 0);
int sl =3*(icmp_seq_counter+1);//for test - check if the watchdog make the NEW_PING to shut down
sleep(sl);
```

7. Sending the packet using the sendto() function (see detailed explaination below):

```
int bytes_sent = sendto(sock, packet, ICMP_HDRLEN + datalen, 0, (struct sockaddr *) &dest_in, sizeof(dest_in));
if (bytes_sent == -1) {
    fprintf(stderr, "sendto() failed with error: %d", errno);
    return -1;      maorsaadon, 4 days ago • wip …
}
```

8. Clearing 'packet' char array and calling the recvfrom function is to receive a packet from the host. And close the time.

```
// Get the ping response
bzero(packet, IP_MAXPACKET);
socklen_t len = sizeof(dest_in);
ssize_t bytes_received = -1;
while ((bytes_received = recvfrom(sock, packet, sizeof(packet), 0, (struct sockaddr *) &dest_in, &len))) {
    if (bytes_received > 0) {
        // Check the IP header
        struct iphdr *iphdr = (struct iphdr *) packet;
        struct icmphdr *icmphdr = (struct icmphdr *) (packet + (iphdr->ihl * 4));
        break;
    }
}

gettimeofday(&end, 0);
```

9. Finally, the round-trip time is calculated and printed:

```
gettimeofday(&end, 0);

//calculate the time
float milliseconds = (end.tv_sec - start.tv_sec) * 1000.0f + (end.tv_usec - start.tv_usec) / 1000.0f;
printf("%ld bytes from %s: icmp_seq=%d ttl=10 time=%f ms)\n", bytes_received, argv[1], icmp_seq_counter++,
       milliseconds);
```

10. closing the socket, and waiting for the child process to finish.

```
// Close the raw socket descriptor.
close(sock);


wait(&status);// waiting for child to finish before exiting
return 0;
```

watchdog.c:

1. WATCHDOG_PORT is defined as 3000:

```
#define WATCHDOG_PORT 3000
```

This is the port number that the watchdog program will listen on.

2. The main() function is then defined, which is the entry point of the program:

```
int main()
{
    // Open the listening socket
    int listeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listeningSocket == -1) {
        printf("Could not create listening socket : %d\n", errno);
        return 1;
    }
```

The socket() function is called to create a new socket.

3. The following code enables the reuse of the port on the listening socket:

```
//fo reusing of port
int enableReuse = 1;
int ret = setsockopt(listeningSocket, SOL_SOCKET, SO_REUSEADDR, &enableReuse, sizeof(int));
if (ret < 0) {
    printf("setsockopt() failed with error code : %d\n", errno);
    return 1;
}
```

The setsockopt() function is called with the SO_REUSEADDR option to allow the reuse of the port on the listening socket. This is useful if the program needs to bind to a port that is still in the process of being released by the operating system after the program previously exited. The enableReuse variable is set to 1 to enable the reuse of the port, and the size of this variable is passed as the fifth argument. If the setsockopt() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

4. The following code creates a sockaddr_in structure to hold the server address and port, and binds the listening socket to this address:

```
struct sockaddr_in serverAddress;
memset(&serverAddress, 0, sizeof(serverAddress));
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = INADDR_ANY;
serverAddress.sin_port = htons(WATCHDOG_PORT);

// Bind the socket to the port with any IP at this port
int bindResult = bind(listeningSocket, (struct sockaddr *) &serverAddress, sizeof(serverAddress));
if (bindResult == -1) {
    printf("Bind failed with error code : %d\n", errno);
    close(listeningSocket);
    return -1;
} else printf("executed Bind() successfully\n");
```

The memset() function is called to clear the serverAddress structure with zeros. The sin_family field is set to AF_INET to specify that the address is an IPv4 address, the sin_addr field is set to INADDR_ANY to specify that the socket should listen on any available IP address, and the sin_port field is set to the WATCHDOG_PORT macro value, converted to network byte order with the htons() function.
The bind() function is then called to bind the listening socket to the serverAddress structure. If the bind() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

5. The following code starts listening for incoming connections on the listening socket:

```
// Make the socket listen.
int listenResult = listen(listeningSocket, 3);
if (listenResult == -1) {
    printf("listen() failed with error code : %d\n", errno);
    close(listeningSocket);
    return -1;
} else printf("Waiting for incoming TCP-connections...\n");
```

The listen() function is called with a backlog of 3 to specify the maximum number of pending connections that the operating system should allow. If the listen() function fails,

it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

6. The following code accepts an incoming connection and creates a new client socket:

```
// Accept and incoming connection
struct sockaddr_in clientAddress;
memset(&clientAddress, 0, sizeof(clientAddress));
socklen_t len_clientAddress = sizeof(clientAddress);
int clientSocket = accept(listeningSocket, (struct sockaddr *) &clientAddress, &len_clientAddress);
if (clientSocket == -1) {
    printf("listen failed with error code : %d\n", errno);
    close(listeningSocket);
    return -1;
} else printf("Connection accepted\n");
```

The accept() function is called to accept an incoming connection and create a new client socket. The clientAddress structure is used to hold the client's address and port, and the len_clientAddress variable is used to store the size of this structure. If the accept() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

7. The following code sets both the listening socket and the client socket to non-blocking mode:

```
//Change the sockets into non-blocking state
fcntl(listeningSocket, F_SETFL, O_NONBLOCK);
fcntl(clientSocket,F_SETFL, O_NONBLOCK);//Change the socket into non-blocking state
```

The fcntl() function is called with the F_SETFL command to set the file descriptor flags for both the listening socket and the client socket. The O_NONBLOCK flag is passed to set the sockets to non-blocking mode. This means that the sockets will not block when the recv() function is called later in the program, allowing the program to continue execution if there is no data available to be received.

8. The following code initializes the start and end timeval structures, and declares the seconds variable:

```
//for calculate the time
struct timeval start, end;
float seconds = 0;
```

The start and end structures will be used to store the start and end times of the program, and the seconds variable will be used to hold the elapsed time between the start and end times.

9. The following code enters a loop where it waits for a message from the client:

```
    //checking if watchdog receive something from ping if yes then update the start time if no then the time
will stop updating and then second>10 -> shut down
    while (seconds <= 10){
        int new_ping_ready = 1;
        int r= recv(clientSocket, &new_ping_ready, sizeof(int), 0);
        if(r>0){
            gettimeofday(&start, 0);
        }
        gettimeofday(&end, 0);
        seconds = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1000000.0f;
    }
```

The recv() function is called to receive a message from the client. If the recv() function returns a value greater than zero, it means that a message was received, and the start time is reset using the gettimeofday() function. The end time is then also retrieved using gettimeofday(), and the elapsed time between the start and end times is calculated and stored in the seconds variable. If the elapsed time is greater than 10 seconds, the loop breaks.

10. The following code closes the client socket and the listening socket:

```
close(clientSocket);//close the clientSocket
close(listeningSocket);//listeningSocket
```

The close() function is called to close both the client socket and the listening socket. This releases the file descriptors associated with the sockets and allows them to be reused.

11. The following code shuts down the system:

```
//reboot the system
kill(getppid(),SIGKILL);
```

In order to finish the program when the timer greater than 10 seconds we use the kill() function sends a signal to a process or process group specified by pid.

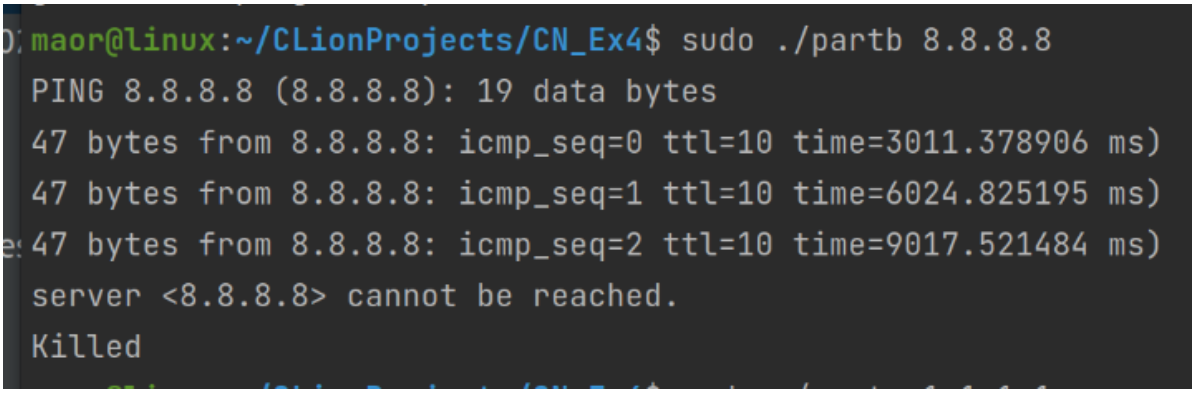
1.2.2 Output

Part a:

Sending a ping massage until the program us terminated by the user (^c):

Part b:

Timing the time between each ICMP request and response, and announcing when the server is unreachable (over 10 secs to respond)



1.2.3 Functions

ping.c:

socket():

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

creates a new socket. Takes three arguments: the domain of the socket (e.g., AF_INET for IPv4, AF_INET6 for IPv6), the type of the socket (e.g., SOCK_STREAM for a stream socket, SOCK_DGRAM for a datagram socket), and the protocol to be used (e.g., 0 for the default protocol). It returns a socket descriptor that can be used to identify the socket in subsequent function calls, or -1 if the socket cannot be created.

strcpy() :

char *strcpy(char *dest, const char *src);

Copies the contents of a null-terminated string pointed to by src to the memory location pointed to by dest. It returns a pointer to dest.

memset():

void *memset(void *s, int c, size_t n);

Fills a block of memory with a specified value. It takes three arguments: a pointer to the memory location to be filled, the value to be written, and the number of bytes to be written. It returns a pointer to the memory location.

inet_addr():

in_addr_t inet_addr(const char *cp);

Converts a string representation of an IPv4 address in dot-decimal notation to a 32-bit binary number in network byte order. It returns the binary representation of the address if the conversion is successful, and INADDR_NONE if the conversion fails.

memcpy():

void *memcpy(void *dest, const void *src, size_t n);

Copies a block of memory from one location to another. It takes three arguments: a pointer to the destination memory location, a pointer to the source memory location, and the number of bytes to be copied. It returns a pointer to the destination memory location.

gettimeofday():

int gettimeofday(struct timeval *tv, struct timezone *tz);

Gets the current time of day. It takes two arguments: a pointer to a struct timeval, which will be filled with the current time, and a pointer to a struct timezone, which will be filled with the current timezone. It returns 0 on success and -1 on failure.

sleep():

Causes the calling process to suspend execution for a specified number of seconds. It takes a single argument: the number of seconds to sleep.

sendto():

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,

const struct sockaddr *dest_addr, socklen_t addrlen);

Sends a message to a socket. It takes six arguments: a socket descriptor, a pointer to the message to be sent, the length of the message, a set of flags, a pointer to a struct sockaddr, and the length of the struct sockaddr. It returns the number of bytes sent on success and -1 on failure.

bzero():

void bzero(void *s, size_t n);

Sets all the bytes in a block of memory to zero. It takes two arguments: a pointer to the memory location to be filled with zeros, and the number of bytes to be set to zero. It returns no value.

recvfrom():

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,

struct sockaddr *src_addr, socklen_t *addrlen);

recvfrom() places the received message into the buffer buf. The caller must specify the size of the buffer in len.
If src_addr is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by src_addr. In this case, addrlen is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with src_addr. Upon return, addrlen is updated to contain the actual size of the source address. The returned address is truncated if the buffer

provided is too small; in this case, addrlen will return a value greater than was supplied to the call.
If the caller is not interested in the source address, src_addr and addrlen should be specified as NULL.

calculate_checksum(unsigned short *paddress, int len):

This function calculates the checksum of a given input buffer: paddress, of length: len. It does this by performing a series of bitwise operations on the data in the buffer.

validateNumber(char *str):

This function checks if the input string: str, consists of only digits. If it does, it returns 1, otherwise it returns 0.

validateIp(char *ip):

This function checks if the input string: ip, is a valid IP address. It does this by splitting the string by the '.' delimiter and checking if each part consists of only digits and is between 0 and 255. If all these conditions are met, it returns 1, otherwise it returns 0.

close():

int close(int fd);

The close function is used to close a file descriptor, such as a socket descriptor. It takes a single argument, which is the file descriptor to be closed. This releases any resources associated with the socket and prevents any further communication through the socket.

better_ping.c:

1. All the functions in ping.c (see above)
2. Additional functions:

fork():

pid_t fork(void);

creates a new process by duplicating the calling process. The new process is called the child process, and the calling process is called the parent process. The child process is an exact copy of the parent process, except for the return value of the fork function. The fork function returns the process ID of the child process to the parent process, and 0 to the child process.

execvp():

int execvp(const char *file, char *const argv[]);

replaces the current process image with a new process image. It takes two arguments: the name of the file to be executed and an array of pointers to null-terminated strings that represent the arguments to the new process. The last element of the array must be a null pointer. It returns no value, but if the execution of the new process image is successful, the calling process is replaced by the new process and does not return. If the execution fails, execvp returns -1 and sets the global variable errno to indicate the error.

wait():

pid_t wait(int *wstatus);

waits for a child process to change its state. It takes two arguments: a pointer to a pid_t variable that will be filled with the process ID of the terminated child process, and an optional pointer to a status variable that will be filled with the exit status of the terminated child process. It returns the process ID of the terminated child process, or -1 if there are no child processes or if an error occurs and sets the global variable errno to indicate the error.

watchdog.c:

1. socket(), memset(), fcntl(), sleep(), gettimeofday() close() (see detailed explanation above).
2. Additional functions:

Bind():

int bind(int sockfd, const struct sockaddr *addr,

socklen_t addrlen);

When a socket is created with socket(), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called "assigning a name to a socket".
It is normally necessary to assign a local address using bind() before a SOCK_STREAM socket may receive connections.
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Listen():

int listen(int sockfd, int backlog);
listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept().
The sockfd argument is a file descriptor that refers to a socket of type SOCK_STREAM or SOCK_SEQPACKET.
The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.
If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Accept():

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and

returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.
The argument sockfd is a socket that has been created with socket(), bound to a local address with bind(), and is listening for connections after a listen().
The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned addr is determined by the socket's address family. When addr is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.
The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr; on return it will contain the actual size of the peer address.
The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.
If no pending connections are present on the queue, and the socket is not marked as nonblocking, accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections arenpresent on the queue, accept() fails with the error EAGAIN or EWOULDBLOCK.

setsockopt():

int setsockopt(int sockfd, int level, int optname,

const void *optval, socklen_t optlen);

Used to set options on a socket. It takes three arguments: the socket descriptor, the level at which the option is defined, and the option name. It can be used to modify a variety of options, including socket timeouts, buffer sizes, and the type of service provided.

fcntl():

int fcntl(int fd, int cmd, ... /* arg */);

Performs file control operations on a file descriptor. It takes three arguments: the file descriptor on which to perform the operation, the command to be executed, and an optional argument that depends on the command. It returns a result that depends on the command, or -1 if the command fails and sets the global variable errno to indicate the error.

kill():

kill [options] <pid> [...]

The kill() function sends a signal to a process or process group specified by pid. The signal to be sent is specified by sig and is either 0 or one of the signals from the list in the <sys/signal. h> header file. The process sending the signal must have appropriate authority to the receiving process or processes.

2 Research findings

2.1 Wireshark

Part a:

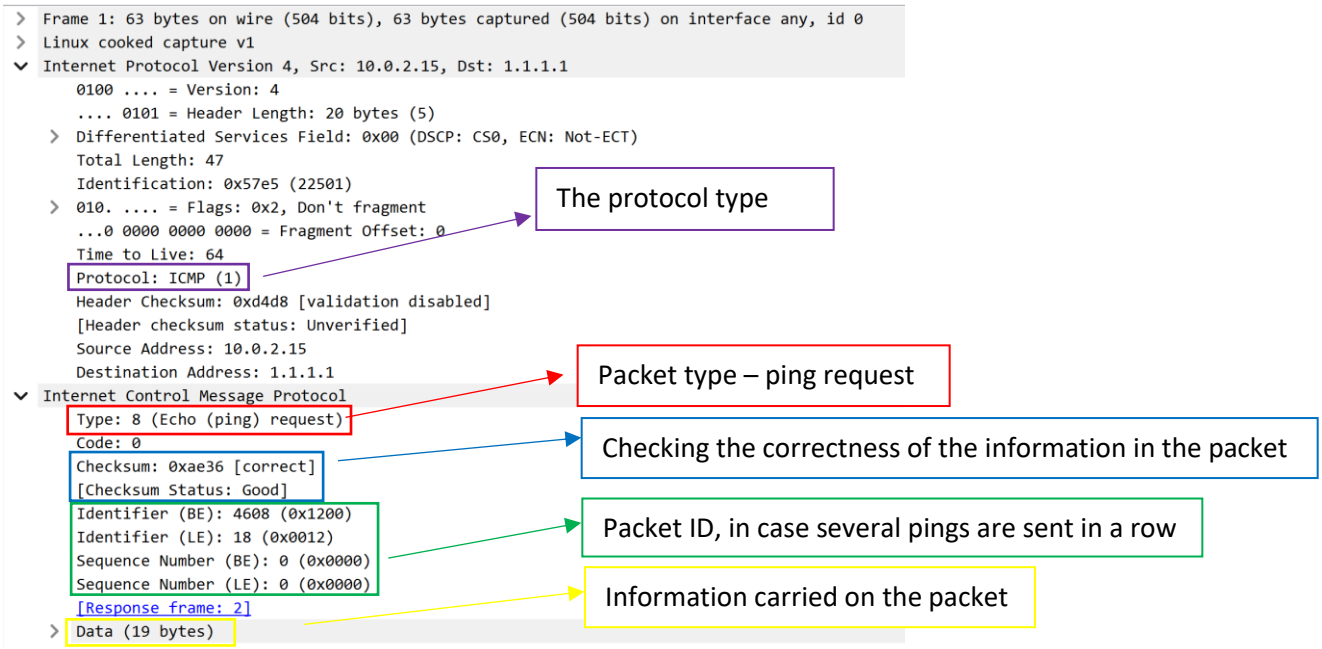
When we put <IP> = 1.1.1.1 on part A:

Notice that we have a lot of ping request and ping reply.

```
 1 0.0… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 2)
 2 0.0… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 1)
 3 1.0… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 4)
 4 1.0… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 3)
 5 2.0… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 6)
 6 2.0… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 5)
 7 3.0… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 8)
 8 3.0… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 7)
 9 4.0… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 10)
10 4.0… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 9)
11 5.0… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 12)
12 5.1… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 11)
13 6.1… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 14)
14 6.1… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 13)
15 7.1… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 16)
16 7.1… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 15)
17 8.1… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 18)
18 8.1… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 17)
19 9.1… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 20)
20 9.2… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 19)
21 10.… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 22)
22 10.… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 21)
23 11.… 10.… 1.1… ICMP          63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 24)
24 11.… 1.1… 10.… ICMP          63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=59 (request in 23)
25 14.… 10.… 208… DNS           102 Standard query 0x62b2 A connectivity-check.ubuntu.com OPT
26 15.… 208… 10.… DNS           246 Standard query response 0x62b2 A connectivity-check.ubuntu.com A 185.125.190.18 A 185.125.190.48 A 185.125.190.49 A 34.122.121.32 A 35.224.170.84 A 35.232.111.17 A 91.189.91.48 A 91.189.91…
27 15.… 10.… 185… TCP            76 35014 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3232158626 TSecr=0 WS=128
28 15.… 185… 10.… TCP            62 80 → 35014 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
29 15.… 10.… 185… TCP            56 35014 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
30 15.… 10.… 185… HTTP          143 GET / HTTP/1.1
31 15.… 185… 10.… TCP            62 80 → 35014 [ACK] Seq=1 Ack=88 Win=65535 Len=0
32 15.… 185… 10.… HTTP          203 HTTP/1.1 204 No Content
33 15.… 10.… 185… TCP            56 35014 → 80 [ACK] Seq=88 Ack=148 Win=64093 Len=0
34 15.… 185… 10.… TCP            62 80 → 35014 [FIN, ACK] Seq=148 Ack=88 Win=65535 Len=0
35 15.… 10.… 185… TCP            56 35014 → 80 [FIN, ACK] Seq=88 Ack=149 Win=64092 Len=0
36 15.… 185… 10.… TCP            62 80 → 35014 [ACK] Seq=149 Ack=89 Win=65535 Len=0
```

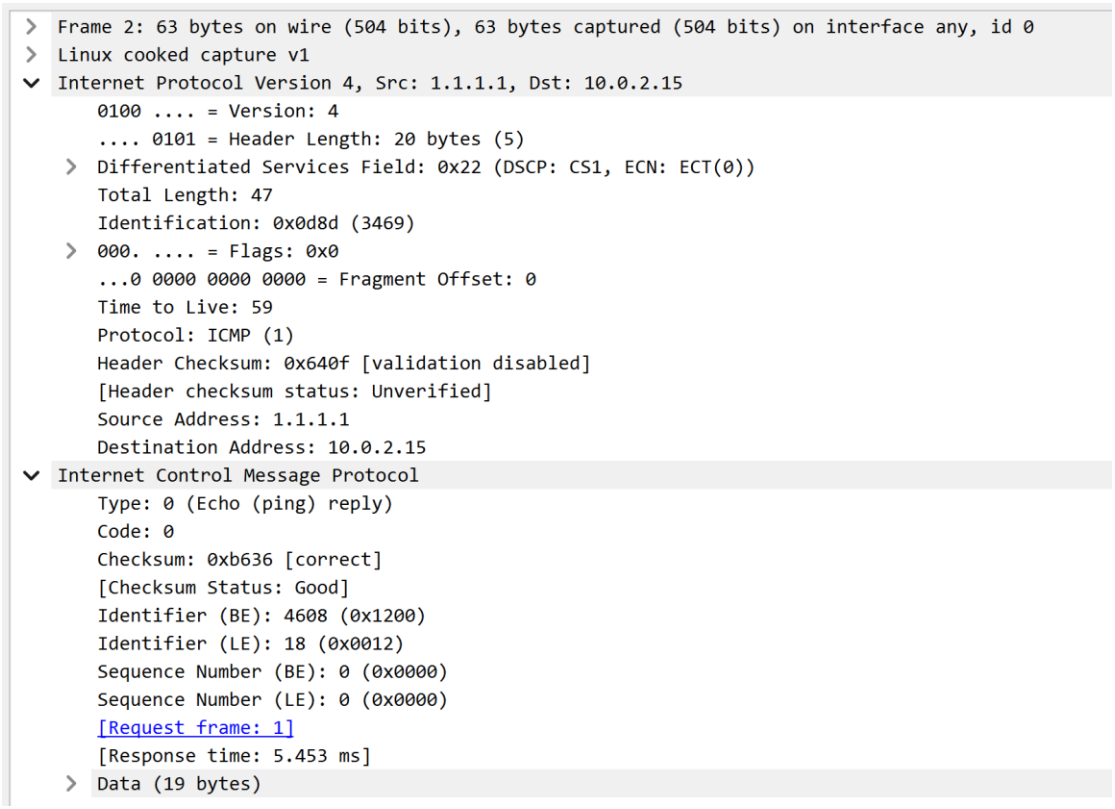
Zoom in:

Ping request:



Ping reply:

We can see that is the same parameters like the ping request.



Part b:

When we put <IP> = 1.1.1.1 on part B

```
    1 0.0… 127… 127… TCP          76 33652 → 3000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=1473224538 TSecr=0 WS=128
    2 0.0… 127… 127… TCP          76 3000 → 33652 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=1473224538 TSecr=1473224538 WS=128
    3 0.0… 127… 127… TCP          68 33652 → 3000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1473224538 TSecr=1473224538
    4 0.0… 127… 127… TCP          72 33652 → 3000 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=1473224539 TSecr=1473224538
    5 0.0… 127… 127… TCP          68 3000 → 33652 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=1473224539 TSecr=1473224539
    6 3.0… 10.… 8.8… ICMP         63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 7)
    7 3.0… 8.8… 10.… ICMP         63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=118 (request in 6)
    8 3.0… 127… 127… TCP          72 33652 → 3000 [PSH, ACK] Seq=5 Ack=1 Win=65536 Len=4 TSval=1473227550 TSecr=1473224539
    9 3.0… 127… 127… TCP          68 3000 → 33652 [ACK] Seq=1 Ack=9 Win=65536 Len=0 TSval=1473227550 TSecr=1473227550
   10 9.0… 10.… 8.8… ICMP         63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 11)
   11 9.0… 8.8… 10.… ICMP         63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=118 (request in 10)
   12 9.0… 127… 127… TCP          72 33652 → 3000 [PSH, ACK] Seq=9 Ack=1 Win=65536 Len=4 TSval=1473233575 TSecr=1473227550
   13 9.0… 127… 127… TCP          68 3000 → 33652 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=1473233575 TSecr=1473233575
   14 13.… 140… 10.… TCP          62 443 → 46412 [ACK] Seq=1 Ack=1 Win=65535 Len=0
   15 18.… 10.… 8.8… ICMP         63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 16)
   16 18.… 8.8… 10.… ICMP         63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=118 (request in 15)
   17 18.… 127… 127… TCP          72 33652 → 3000 [PSH, ACK] Seq=13 Ack=1 Win=65536 Len=4 TSval=1473242593 TSecr=1473233575
   18 18.… 127… 127… TCP          68 3000 → 33652 [ACK] Seq=1 Ack=17 Win=65536 Len=0 TSval=1473242593 TSecr=1473242593
   19 28.… 127… 127… TCP          68 3000 → 33652 [FIN, ACK] Seq=1 Ack=17 Win=65536 Len=0 TSval=1473252595 TSecr=1473242593
   20 28.… 127… 127… TCP          68 33652 → 3000 [FIN, ACK] Seq=17 Ack=2 Win=65536 Len=0 TSval=1473252595 TSecr=1473252595
   21 28.… 127… 127… TCP          68 3000 → 33652 [ACK] Seq=2 Ack=18 Win=65536 Len=0 TSval=1473252595 TSecr=1473252595
```

Zoom in:

Open TCP connection:

```
 1 0.0… 127… 127… TCP          76 33652 → 3000 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=1473224538 TSecr=0 WS=128
 2 0.0… 127… 127… TCP          76 3000 → 33652 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=1473224538 TSecr=1473224538 WS=128
```

we can see the ack sign (int : len = 4) that the better_ping sent to watchdog for restart the timer. Then we can see the ping request an ping reply.

```
 3 0.0… 127… 127… TCP          68 33652 → 3000 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=1473224538 TSecr=1473224538
 4 0.0… 127… 127… TCP          72 33652 → 3000 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=1473224539 TSecr=1473224538
 5 0.0… 127… 127… TCP          68 3000 → 33652 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=1473224539 TSecr=1473224539
 6 3.0… 10.… 8.8… ICMP         63 Echo (ping) request  id=0x1200, seq=0/0, ttl=64 (reply in 7)
 7 3.0… 8.8… 10.… ICMP         63 Echo (ping) reply    id=0x1200, seq=0/0, ttl=118 (request in 6)
```

Closing the TCP connection when the ICMP message take more then 10 seconds

```
19 28.… 127… 127… TCP          68 3000 → 33652 [FIN, ACK] Seq=1 Ack=17 Win=65536 Len=0 TSval=1473252595 TSecr=1473242593
20 28.… 127… 127… TCP          68 33652 → 3000 [FIN, ACK] Seq=17 Ack=2 Win=65536 Len=0 TSval=1473252595 TSecr=1473252595
```

2.5 Bibliography

Ping

https://www.geeksforgeeks.org/ping-in-c/

For checking the IP

https://www.tutorialspoint.com/c-program-to-validate-an-ip-address

Watchdog

https://www.youtube.com/watch?v=xJb-btYIYYA&ab_channel=udemyjobzzsolutions

Fork

https://www.geeksforgeeks.org/fork-system-call/

About the functions
'man' command on vs terminal

For many things

ChatGPT: Optimizing Language Models for Dialogue (openai.com)

https://www.google.com/search?gs_ssp=eJzj4tTP1TcwMU02T1JgNGB0YPBiS8_PT89JBQBASQXT&q=google&oq=googlr&aqs=chrome.1.69i57j46i10i131i199i433i465i512j0i10i131i433i512l4j69i60j69i65.3825j0j4&sourceid=chrome&ie=UTF-8