Prof. Nicola Zingirian - Computer Networks -

EXPERIENCE 1

This program is designed to work with IPv4 addresses and uses the standard C library for network communication and input/output operations. It will connect to a server, send a simple HTTP GET request for the root path ("/"), and print the response.

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

#define PORT 80
#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    int sockfd, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[BUFFER_SIZE];

    // Check if the hostname is provided
    if (argc < 2) {
        fprintf(stderr, "usage %s hostname\n", argv[0]);
        exit(0);
    }

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("ERROR opening socket");
        exit(1);
    }

    // Resolve the hostname to an IP address
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }

    // Print the resolved IP address
    printf("%d.%d.%d.%d\n",
           ((unsigned char*)server->h_addr)[0],
           ((unsigned char*)server->h_addr)[1],
           ((unsigned char*)server->h_addr)[2],
```

```
            ((unsigned char*)server->h_addr)[3]);

    // Fill in the server's address structure
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    memcpy(&serv_addr.sin_addr.s_addr, server->h_addr,
server->h_length);
    serv_addr.sin_port = htons(PORT);

    // Connect to the server
    if (connect(sockfd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) < 0) {
        perror("ERROR connecting");
        exit(1);
    }

    // Send the HTTP GET request
    sprintf(buffer, "GET / HTTP/1.1\r\nHost: %s\r\nConnection:
close\r\n\r\n", argv[1]);
    n = write(sockfd, buffer, strlen(buffer));
    if (n < 0) {
        perror("ERROR writing to socket");
        exit(1);
    }

    // Read and print the server's response
    memset(buffer, 0, BUFFER_SIZE);
    while ((n = read(sockfd, buffer, BUFFER_SIZE-1)) > 0) {
        printf("%s", buffer);
        memset(buffer, 0, BUFFER_SIZE);
    }
    if (n < 0) {
        perror("ERROR reading from socket");
        exit(1);
    }

    // Clean up and close the socket
    close(sockfd);
    return 0;
}
```

To compile this program, you can use the `gcc` compiler:

```
gcc -o http_request http_request.c
```

And then run it, passing a hostname as an argument:

```
./http_request example.com
```

This program makes a basic effort to interact with a web server but lacks many features and robustness checks you'd find in real-world software, such as handling HTTPS, redirect responses, or chunked transfer encoding.

Remember, this example is for educational purposes and demonstrates the low-level mechanics of network programming in C. In production code, you would likely use a more comprehensive approach or a library designed for HTTP requests to handle the complexity and variability of network communications more effectively.

Creating a simple HTTP GET request in Python is much more straightforward than in C, thanks to Python's high-level abstractions and the `socket` library that is part of the standard library. Below is a Python script that performs a basic HTTP GET request to a specified host and prints the response. This script uses Python's `socket` module to establish a TCP connection, send the HTTP request, and receive the response.

```
import socket

def http_get(host, path="/"):
 request = f"GET {path} HTTP/1.1\r\nHost: {host}\r\nConnection: close\r\n\r\n"
 buffer_size = 4096

 # Resolve host
 host_ip = socket.gethostbyname(host)

 # Create socket
 with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
 s.connect((host_ip, 80))
 s.send(request.encode())

 # Receive the response
 response = ""
 while True:
 part = s.recv(buffer_size)
 if not part:
 break
 response += part.decode()

 return response

if __name__ == "__main__":
 host = "example.com"
 print(http_get(host))
```

This script defines a `http_get` function that takes a host and an optional path as arguments. It constructs an HTTP GET request string, resolves the host name to an IP address, establishes a TCP connection to the host on port 80, sends the request, and then receives the response in chunks until there's nothing left to read.

To run this script, save it to a file (e.g., `http_request.py`) and execute it with Python:

```
python http_request.py
```

This Python script handles the basics of making an HTTP GET request and can be easily modified or extended for more complex requests. However, even here, for real-world applications or more feature-complete HTTP requests (including handling HTTPS, redirects, etc.), it's recommended to use higher-level libraries such as `requests` in Python, which greatly simplify HTTP communication with a more intuitive interface.

In the Python code example provided for making an HTTP GET request using the `socket` module, the `encode()` and `decode()` methods are used to convert strings (in this case, the HTTP request and the response) between bytes and string (text) formats. Here's how they're used and what they represent:

encode()

The `encode()` method in Python converts a string into bytes, using a specified encoding scheme. If no encoding is specified, it defaults to UTF-8. This conversion is necessary because network communication happens in bytes, and the `socket` module's `send()` function requires data to be in byte format.

In the HTTP request construction:

```
request = f"GET {path} HTTP/1.1\r\nHost: {host}\r\nConnection: close\r\n\r\n"
s.send(request.encode())
```

The `request` string is encoded to bytes using the `encode()` method before it's sent over the network with `s.send()`.

decode()

Conversely, the `decode()` method converts bytes back into a string using a specified encoding scheme, which also defaults to UTF-8 if not specified. This method is used because the response

from the server comes in bytes, and for it to be readable or processed as text in Python, it needs to be converted back to a string format.

In the response handling part:

```
part = s.recv(buffer_size)
response += part.decode()
```

The bytes received from the server (`part`) are decoded back into a string using `decode()`, allowing the `response` to be concatenated as a readable string.

Summary

- `encode()`: String → Bytes. Used to prepare text data for transmission or storage that requires byte format.
- `decode()`: Bytes → String. Used to convert received or stored byte data back into a string format for processing or display.

These methods are essential for handling text in binary formats, such as when working with files, data transmission, or when interfacing with APIs or network services that communicate using text-based protocols like HTTP.

Exercise

Focusing the comparison on network functions and performance between a C and a Python program using `strace` with the `-ttt` option can provide insights into the timing and efficiency of network-related system calls. The `-ttt` option in `strace` prints an absolute timestamp in microseconds since the Epoch (1970-01-01), which is valuable for analyzing the performance.

Task Simplification

For this exercise, you will still compare the `strace` outputs of the C and Python HTTP request programs, but with a narrowed focus on:

- Network-related system calls (`socket()`, `connect()`, `send()`, `recv()`, `close()`).
- The timing and duration of these calls to assess performance.

Steps to Follow

Run `strace` with `-ttt` Option:

Execute both programs with `strace`, using the `-ttt` option for precise timing, and redirect the outputs to separate files.

- For the compiled C program:

```
strace -ttt -o strace_c_network.txt ./http_request_c example.com
```

- For the Python script:

```
strace -ttt -o strace_python_network.txt python3 http_request.py
```

Analyze the Timing of Network Operations:

- Start and End Times: For each network-related system call, note the timestamp at the start and the completion. This will help you calculate the duration of each call.
- Comparison: Compare the start, end, and duration of similar network operations between the C and Python programs. Pay particular attention to potentially longer waits or overheads in one implementation versus the other.

Performance Analysis:

- Latency: Identify any significant differences in latency for establishing a connection (`connect()`), sending (`send()`), and receiving data (`recv()`).
- System Call Efficiency: Assess which program makes more efficient use of network-related system calls in terms of speed and the number of calls made.

Hints for Analysis

- Timestamps: Use the absolute timestamps to calculate the exact duration of each network-related system call. This can reveal which parts of the network communication are slower or potentially optimized.
- System Calls Count: While focusing on network functions, also consider the total count of these calls as an indicator of efficiency. More system calls might suggest additional overhead.
- Error Handling and Retries: Look for retries or error handling in network operations, as these can impact performance and reveal differences in how each program manages network unreliability.

Reporting

Your report should include:

- A table or list summarizing the start, end, and duration of key network-related system calls for both programs.
- Observations on the performance differences between the C and Python implementations, with a focus on latency and efficiency of network operations.
- Discussion on any patterns or anomalies observed in the timing of system calls, including potential reasons for significant delays or faster execution in one program versus the other.
- Conclusions on how the language and its runtime environment (e.g., C vs. Python interpreter) impact the performance of simple network operations.

This focused comparison will help you understand not just how different programming languages handle network operations at a system call level, but also how these differences affect the performance and efficiency of network communication in real-world applications

EXPERIENCE 2

To illustrate how the C language serves as an interface not just to the programmer but also to the computer's architecture, particularly regarding memory access, we'll explore various examples. These examples will demonstrate how C allows direct interaction with memory, adherence to computer architecture's principles, and manipulation of data at a low level.

C provides a much closer level of control over RAM and how memory is allocated, used, and freed compared to higher-level languages like Python. In C, programmers are responsible for directly managing memory, which includes allocating it dynamically, using it to store data, and freeing it when it's no longer needed. This direct memory management is powerful but requires careful handling to avoid leaks or corruption.

In C, the `printf` function uses format specifiers to interpret the raw bits of data in memory according to the data type intended by the programmer. This illustrates the distinction between the data's raw binary representation and its meaning (as an integer, float, character, etc.).

Here's a brief overview of common `printf` format specifiers and how they relate to data meaning and raw representation:

- *`%d` or `%i`: Interpret as a signed integer.*
- *`%u`: Interpret as an unsigned integer.*
- *`%f`: Interpret as a floating-point number. This expects the corresponding argument to be of type `double` (or `float` auto-promoted to `double`).*
- *`%c`: Interpret as a single character.*
- *`%s`: Interpret as a string (null-terminated array of characters).*
- *`%p`: Interpret as a pointer address.*
- *`%x` or `%X`: Interpret as a hexadecimal representation of an unsigned integer.*
- *`%o`: Interpret as an octal representation of an unsigned integer.*
- *`%e` or `%E`: Interpret as a scientific notation (e.g., 1.2e+2 for 120).*

- `%g` or `%G`: Interpret in either `%f` or `%e` format, whichever is more compact, without insignificant zeros.
- `%ld`, `%lu`, `%llx`, etc.: Variants for different sizes of integers (`long`, `long long`, `unsigned`, etc.).

Each specifier tells `printf` how to interpret the bits of the corresponding argument: as an integer, floating-point number, character, etc. This distinction is crucial for correctly displaying or processing raw data, as the same set of bits can represent vastly different values depending on the intended interpretation. For example, the bit pattern `01000001` could represent the integer `65`, the floating-point number `3.25E-44` (in a very rough, illustrative sense, due to how floating-point is encoded), or the character `'A'` in ASCII, depending on how those bits are interpreted.

This direct control and explicit interpretation of memory are what make C powerful for low-level programming, allowing for efficient manipulation of data in a way that closely matches the underlying hardware's operations. In contrast, higher-level languages like Python abstract away these details, trading off some efficiency and control for ease of use and safety.

Accessing Memory Addresses

In C, every variable is stored at a specific memory address, which can be accessed using pointers. Here's a simple example to demonstrate this:

```c
#include <stdio.h>

int main() {
    int x = 10;
    int y = 20;

    printf("Address of x: %p\n", (void*)&x);
    printf("Address of y: %p\n", (void*)&y);

    return 0;
}
```

This program prints the memory address of the variable `x`. The use of `&x` obtains the address, and `%p` in the `printf` statement formats it as a pointer address.

Array Indexing and Memory Displacement

C arrays are closely tied to memory. The array index corresponds to the displacement from the array's starting address:

```c
#include <stdio.h>

int main() {
 int arr[5] = {10, 20, 30, 40, 50};
 printf("Address of arr[0]: %p\n", (void*)&arr[0]);
 printf("Address of arr[1]: %p\n", (void*)&arr[1]);
 return 0;
}
```

The difference in addresses between `arr[1]` and `arr[0]` corresponds to the size of an `int` on the machine, demonstrating how indexing reflects memory displacement.

Correspondence with CPU Data Types

C's basic data types are designed to match the data types supported by the CPU:

```c
#include <stdio.h>

int main() {
 printf("Size of char: %zu bytes\n", sizeof(char));
 printf("Size of int: %zu bytes\n", sizeof(int));
 printf("Size of float: %zu bytes\n", sizeof(float));
 printf("Size of double: %zu bytes\n", sizeof(double));
 return 0;
```

This program outputs the sizes of basic data types, which are directly mapped to the CPU's capabilities to handle data of different sizes and types[1].

It is possible to create a correspondence table between basic C data types, their standard library equivalents in `<stdint.h>`) and typical sizes on most today's platforms :

- An `int` is typically 32 bits (4 bytes).
- A `short` is typically 16 bits (2 bytes).
- A `long` can be 32 bits on 32-bit systems or 64 bits on 64-bit systems (4 or 8 bytes respectively).

[1] The `%zu` format specifier in C is used with the `printf` function (or similar functions) for printing `size_t` types. `size_t` is an unsigned integer data type used to represent sizes of objects or indices in arrays, and it is returned by the `sizeof` operator. The `size_t` type is defined in the `<stddef.h>` (and other) header files and is guaranteed to be large enough to contain the size in bytes of the largest object the host system can handle. This makes it portable and safe for memory sizes, avoiding the risks of integer overflow when working with large sizes or a large number of elements.

- `%z` is a length modifier that specifies the size of the `size_t` or `ssize_t` type.
- `u` stands for unsigned decimal integer.

Therefore, `%zu` is used to print a `size_t` value as an unsigned decimal integer. The use of `%zu` ensures that the program will be portable across platforms with different sizes of `size_t`, which might be 32 bits on some systems and 64 bits on others.

- A `long long` is typically 64 bits (8 bytes).
- A `char` is 1 byte (8 bits), and it is the smallest addressable unit in C.

Please note that the actual size of these types can vary depending on the architecture, compiler, and even compiler flags used. However, the `<stdint.h>` header provides fixed-width integers with precise sizes, improving portability across different platforms.

Basic C Type	Standard Type (`<stdint.h>`)	Typical Size (bytes)	Description
`char`	`int8_t` / `uint8_t`	1	Signed or unsigned 8-bit integer
`short`	`int16_t` / `uint16_t`	2	Signed or unsigned 16-bit integer
`int`	`int32_t` / `uint32_t`	4	Signed or unsigned 32-bit integer
`long`	`int32_t` / `uint32_t` *or* `int64_t` / `uint64_t`	4 or 8	Signed or unsigned integer (platform-dependent size)
`long long`	`int64_t` / `uint64_t`	8	Signed or unsigned 64-bit integer

It's important to mention a few nuances:

- The types `int8_t`, `int16_t`, `int32_t`, and `int64_t` (and their unsigned counterparts) are guaranteed to have exactly 8, 16, 32, and 64 bits, respectively. If a platform cannot provide an integer of exactly that width with no padding bits, it will not define that type.
- The `long` type's size varies notably between systems: it's commonly 32 bits on 32-bit systems and 64 bits on 64-bit systems.
- The `long double` size is highly compiler and architecture-dependent, ranging widely in size and precision.

This table is a simplified overview intended for typical cases. For precise, platform-specific sizes, it's best to use the `sizeof` operator or consult specific compiler documentation.

Struct Packing and Alignment

Structs in C are packed in memory in a predictable manner, although padding may be added for alignment. This alignment is crucial for performance on many architectures:

```c
#include <stdio.h>

typedef struct {
 char a; // 1 byte
 int b; // 4 bytes, might have padding before it for alignment
} Example;

int main() {
 printf("Size of Example: %zu bytes\n", sizeof(Example));
 return 0;
```

This example may show that the size of `Example` is more than the sum of its parts due to alignment requirements.

Exercise: Endianness Detection

Endianness refers to the order in which bytes are arranged within a larger data type. Here's a simple program to detect if a system is little-endian or big-endian:

```c
#include <stdio.h>

int main() {
 unsigned int x = 1;
 char *c = (char*)&x;
 if (*c)
 printf("Little-endian\n");
 else
 printf("Big-endian\n");
 return 0;
}
```

This program uses a character pointer to inspect the first byte of an integer. If the first byte contains the least significant byte of the integer (`1`), the system is little-endian. Otherwise, it's big-endian.

These examples collectively demonstrate how C acts as an interface to the computer's architecture, allowing programmers to manipulate memory and data types in a manner that's both flexible and closely aligned with the underlying hardware.

To illustrate accessing the `y` variable using a pointer starting from the `x` variable, we'll need to set up a scenario where `x` and `y` are adjacent in memory, typically in an array or struct. To clarify and align with the request, we'll use a struct containing two integers, `x` and `y`. Then, we'll demonstrate how to access `y` using a pointer to `x`.

Here's an example demonstrating this concept:

```c
#include <stdio.h>

typedef struct {
 int x;
 int y;
} XY;

int main() {
 XY xy = {10, 20}; // Initialize x to 10 and y to 20
 int *p = &xy.x; // Pointer to x

 // Access x directly via pointer
 printf("Value of x: %d\n", *p);

 // Access y by moving the pointer to the next integer position
 printf("Value of y: %d\n", *(p + 1));

 return 0;
}
```

In this example, `p` is a pointer to an integer, initially pointing to `x` within our structure `XY`. Since `x` and `y` are adjacent in memory within the structure, moving the pointer to the next integer position

(`p + 1`) points it to `y`. We then use the dereference operator (`*`) on `(p + 1)` to access the value of `y`.

This technique leverages the understanding of how data is laid out in memory and how pointer arithmetic can be used to navigate through it. It's a clear demonstration of the control C provides over memory, allowing for manipulation at a very granular level.

Differently for local variables like `x` and `y` declared in sequence within a function, the C standard does not guarantee their memory layout in the same way it does for elements within a struct or array. The compiler may arrange local variables in memory as it sees fit, potentially reordering them or inserting padding for optimization purposes.

However, for the purpose of demonstration and under the assumption that a specific compiler and architecture might place these variables adjacently in memory in the order of declaration, as in the case of the cloud platform we can still write an illustrative example. Remember, this behavior is not guaranteed and can vary between compilers, compiler versions, optimization levels, and target architectures. This example is purely educational and should not be relied upon in practical programming:

```
#include <stdio.h>

int main() {
 int x = 10; // Assume x is declared first
 int y = 20; // Assume y is declared next and placed adjacent in memory by the
compiler

 int *px = &x; // Pointer to x

 // WARNING: This operation relies on undefined behavior
 // It assumes y is stored in memory immediately after x
 int *py = px + 1; // Attempt to point to y by advancing px

 // Display values using pointers
 printf("Value of x: %d\n", *px);
 printf("Value of y: %d (via pointer arithmetic, undefined behavior)\n", *py);

 return 0;
```

This example attempts to access `y` by creating a pointer `px` to `x` and then incrementing it to point to `y`. However, it's crucial to emphasize that this approach relies on undefined behavior according to the C standard, as there is no guarantee that `x` and `y` will be placed adjacently in memory in the order they are declared.