Assignment_23-08-2024

Author: ThanhTH10
Date: 26/08/2024

1. copy tcp packet and ip packet to char buffer by using reinterpt cast operation

```cpp
#include <iostream>
#include <cstring>  // For memcpy

// Example structures for IP and TCP packets
struct IPPacket {
    uint8_t version;
    uint8_t headerLength;
    uint16_t totalLength;
    // Other fields...
};
struct TCPPacket {
    uint16_t sourcePort;
    uint16_t destPort;
    uint32_t sequenceNumber;
    // Other fields...
};
int main() {
    // Create an IP packet and a TCP packet
    IPPacket ipPacket = { 4, 20, 40 };  // Example values
    TCPPacket tcpPacket = { 8080, 80, 12345 };  // Example values
    // Allocate a char buffer large enough to hold either packet
    char buffer[1024];
    // Copy the IP packet to the char buffer
    std::memcpy(buffer, reinterpret_cast<char*>(&ipPacket), sizeof(ipPacket));
    std::cout << "IP Packet copied to buffer." << std::endl;
    // Copy the TCP packet to the char buffer
    std::memcpy(buffer, reinterpret_cast<char*>(&tcpPacket), sizeof(tcpPacket));
    std::cout << "TCP Packet copied to buffer." << std::endl;
    return 0;
}
```

2.
struct Data {
int x;
float y;
};

char buffer[sizeof(Data)];
Data* pData = reinterpret_cast<Data*>(buffer); // Treat the char array as a Data struct
write program with formatted packet and show results

```cpp
#include <iostream>
#include <cstring>

struct Data
{
    int x;
    float y;
};
```

```cpp
int main()
{
    // Initialize a Data structure
    Data originalData = {42, 3.14f};
    // Create a char buffer and copy the contents of originalData into it
    char buffer[sizeof(Data)];
    std::memcpy(buffer, &originalData, sizeof(Data));
    // Use reinterpret_cast to treat the buffer as a Data structure
    Data *pData = reinterpret_cast<Data *>(buffer);
    // Display the results
    std::cout << "Original Data: " << std::endl;
    std::cout << "x: " << originalData.x << ", y: " << originalData.y << std::endl;
    std::cout << "Reinterpreted Data from buffer: " << std::endl;
    std::cout << "x: " << pData->x << ", y: " << pData->y << std::endl;
    return 0;
}
```

Output:

```
Original Data:
x: 42, y: 3.14
Reinterpreted Data from buffer:
x: 42, y: 3.14
```

3. Application of below code. Write similar code for I2C register space structure unsigned int ***regptr=reinterpt_cast<unsigned int****>(0x400000)

```cpp
#include <cstdint>

// Define the base address of the I2C register space
constexpr uintptr_t I2C_BASE_ADDR = 0x40005400;
// Define a structure representing the I2C register map
struct I2C_Registers
{
    volatile uint32_t CR1;   // Control register 1
    volatile uint32_t CR2;   // Control register 2
    volatile uint32_t OAR1;  // Own address register 1
    volatile uint32_t OAR2;  // Own address register 2
    volatile uint32_t DR;    // Data register
    volatile uint32_t SR1;   // Status register 1
    volatile uint32_t SR2;   // Status register 2
    volatile uint32_t CCR;   // Clock control register
    volatile uint32_t TRISE; // TRISE register
};
// Create a pointer to the I2C register space
I2C_Registers *i2c = reinterpret_cast<I2C_Registers *>(I2C_BASE_ADDR);
int main()
{
    // Set the CR1 register
    i2c->CR1 = 0x0001;
    // Read the SR1 register
    uint32_t status = i2c->SR1;
    return 0;
}
```

4. Explore about linear data structures and non linear data structures

Linear Data Structures

In linear data structures, elements are arranged sequentially, one after the other. This means each element is connected to its previous and next element in a single level. Here are some common examples:

- **Array**: A collection of elements identified by index or key. All elements are of the same type and stored in contiguous memory locations.
- **Stack**: Follows the Last In, First Out (LIFO) principle. Elements are added (pushed) and removed (popped) from the top.
- **Queue**: Follows the First In, First Out (FIFO) principle. Elements are added (enqueued) at the rear and removed (dequeued) from the front.
- **Linked List**: Consists of nodes where each node contains data and a reference (or link) to the next node in the sequence.

Non-Linear Data Structures

In non-linear data structures, elements are not arranged sequentially. Instead, they are arranged in a hierarchical manner, which allows for more complex relationships between elements. Here are some common examples:

- **Tree**: A hierarchical structure with a root node and child nodes, forming a parent-child relationship. Each node can have multiple children but only one parent.
- **Graph**: Consists of vertices (nodes) and edges (connections) that may form complex networks. Graphs can be directed or undirected, and they can represent various relationships between elements.

5. write program to perform hexa decimal addition with out using 0x, read eleements in decimal

```cpp
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
string decimalToHexadecimal(int n)
{
    stringstream ss;
    ss << hex << n;
    return ss.str();
}

int hexadecimalToDecimal(string hex_string)
{
    int decimal = 0;
    stringstream ss;
    ss << hex << hex_string;
    ss >> decimal;
    return decimal;
}
string addHexadecimalNumbers(string hex_a, string hex_b)
{
    int decimal_a = hexadecimalToDecimal(hex_a);
    int decimal_b = hexadecimalToDecimal(hex_b);
    int sum_decimal = decimal_a + decimal_b;
    return decimalToHexadecimal(sum_decimal);
}
```

```
int main()
{
    int a, b;
    cout << "Enter first decimal number: ";
    cin >> a;
    cout << "Enter second decimal number: ";
    cin >> b;
    // Convert decimal to hexadecimal
    string hex_a = decimalToHexadecimal(a);
    string hex_b = decimalToHexadecimal(b);
    // Perform hexadecimal addition
    string result = addHexadecimalNumbers(hex_a, hex_b);
    cout << "The sum of the two hexadecimal numbers is: " << result << endl;
    return 0;
}
```

Output:

```
Enter first decimal number: 10
Enter second decimal number: 20
The sum of the two hexadecimal numbers is: 1e
```

6. Write program for stack by using character input data

```
#include <iostream>
#include <stack>
using namespace std;

int main(int argc, char const *argv[])
{
    stack<char> char_stack;
    char element;
    cout << "Enter charactor to push onto stack: " << endl;
    while (true)
    {
        cin >> element;
        if (element == '0')
        {
            break;
        }
        char_stack.push(element);
    }
    cout << "-----Stack-------" << endl;
    while (!char_stack.empty())
    {
        cout << char_stack.top() << " ";
        char_stack.pop();
    }
    return 0;
}
```

Output:

```
Enter charactor to push onto stack:
h
e
l
l
o
0
-----Stack-------
o l l e h
```

7. difference between set and multiset

Feature	set	multiset

Uniqueness	Stores only unique elements	Allows duplicate elements
Order	Elements are stored in sorted order	Elements are stored in sorted order
Insertion	Duplicate insertions are ignored	Duplicate insertions are allowed
Implementation	Typically a balanced binary search tree (e.g., Red-Black Tree)	Typically a balanced binary search tree (e.g., Red-Black Tree)
Use Case	When unique elements are required	When duplicates are allowed or needed

8. difference between map and multimap

Feature	map	multimap
Key Uniqueness	Keys must be unique	Allows multiple elements with the same key
Element Access	Supports [] operator for access	Does not support [] operator
Insertion	Replaces value if key already exists	Allows multiple values for the same key
Usage Scenario	When unique keys are needed	When multiple values per key are needed
Example	myMap[2] = "Another Two";	myMultimap.insert({2, "Another Two"});

9. In operating system
-- why we need threads
-- what is shared memory
-- what is mutex
-- what is semaphore

Why We Need Threads

Threads are essential for improving the performance and responsiveness of applications. Here are some key reasons:

- **Concurrency**: Threads allow multiple tasks to run concurrently within a single process. This is especially useful for applications that need to perform multiple operations simultaneously, such as a web server handling multiple client requests.
- **Resource Sharing**: Threads within the same process share the same memory space, which makes it easier to share data and resources between them without the need for complex inter-process communication.
- **Responsiveness**: In GUI applications, threads can keep the user interface responsive by performing long-running tasks in the background.
- **Utilizing Multi-core Processors**: Modern processors have multiple cores. Threads can run on different cores, making better use of the CPU and improving performance.

Shared Memory

Shared memory is a method of inter-process communication (IPC) that allows multiple processes to access the same memory space. This is useful for sharing data between processes without the overhead of copying data between them. Key points include:

- **Efficiency**: Shared memory is one of the fastest IPC mechanisms because it avoids the need to copy data between processes.

- **Synchronization**: Processes need to synchronize access to shared memory to avoid data corruption. This is typically done using synchronization primitives like mutexes and semaphores.

Mutex

A mutex (short for mutual exclusion) is a synchronization primitive used to protect shared resources from concurrent access by multiple threads. Key characteristics include:

- **Exclusive Access**: Only one thread can hold the mutex at a time, ensuring exclusive access to the shared resource.
- **Lock and Unlock**: Threads must lock the mutex before accessing the shared resource and unlock it when done.
- **Preventing Race Conditions**: Mutexes help prevent race conditions, where the outcome depends on the sequence or timing of uncontrollable events.

Semaphore

A semaphore is another synchronization primitive used to control access to shared resources. It can be used for both mutual exclusion and signaling between threads. Key points include:

- **Counting Semaphore**: Maintains a count, which can be incremented or decremented by threads. The count represents the number of available resources.
- **Binary Semaphore**: Similar to a mutex, but can only take values 0 or 1. It is used for signaling purposes.
- **Synchronization**: Semaphores can be used to synchronize the execution of threads, ensuring that certain operations are performed in a specific order.