Assignment_13-08-2024

Author: ThanhTH10
Date: 13/08/2024

---

1. what are function pointers in C, explain 3 real time example scenarios for function pointers.

Function pointer is a variable that holds the address of a function. This allows you to call functions indirectly and can be useful for implementing callbacks, dynamic function dispatch, and other advanced programming techniques.

```c
#include <stdio.h>

// A function that matches the signature of the function pointer
void sayHello() {
    printf("Hello, World!\n");
}
int main() {
    // Declaring a function pointer
    void (*funcPtr)();

    // Assigning the address of the function to the pointer
    funcPtr = sayHello;

    // Calling the function using the function pointer
    funcPtr();

    return 0;
}
```

Real-time scenarios where function pointers

1. Callback Functions for Sorting Algorithms

In sorting algorithms, function pointers are often used to pass comparison functions. This allows the same sorting function to be used with different comparison criteria.

```c
#include <stdio.h>
#include <stdlib.h>

// Comparison function for ascending order
int compareAsc(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}
// Comparison function for descending order
int compareDesc(const void *a, const void *b) {
    return (*(int*)b - *(int*)a);
}
void sortArray(int *arr, size_t size, int (*compare)(const void *, const void *)) {
    qsort(arr, size, sizeof(int), compare);
}
int main() {
    int arrAsc[] = {5, 2, 9, 1, 5, 6};
    int arrDesc[] = {3, 6, 8, 2, 9, 4};

    // Sort in ascending order
    sortArray(arrAsc, sizeof(arrAsc)/sizeof(arrAsc[0]), compareAsc);
    printf("Ascending: ");
    for (int i = 0; i < sizeof(arrAsc)/sizeof(arrAsc[0]); i++) {
        printf("%d ", arrAsc[i]);
    }
```

```
    printf("\n");

    // Sort in descending order
    sortArray(arrDesc, sizeof(arrDesc)/sizeof(arrDesc[0]), compareDesc);
    printf("Descending: ");
    for (int i = 0; i < sizeof(arrDesc)/sizeof(arrDesc[0]); i++) {
        printf("%d ", arrDesc[i]);
    }
    printf("\n");

    return 0;
}
```

2. Event Handlers in an Embedded System

In embedded systems, function pointers are often used to handle different events like timers or interrupts. This allows you to register different functions to handle various events.

```
#include <stdio.h>

typedef void (*EventHandler)();
void timerInterruptHandler() {
    printf("Timer interrupt handled.\n");
}
void buttonPressHandler() {
    printf("Button pressed.\n");
}
void registerEventHandler(EventHandler handler) {
    // Simulate an event happening
    handler();
}
int main() {
    // Register and handle a timer interrupt
    registerEventHandler(timerInterruptHandler);
    // Register and handle a button press
    registerEventHandler(buttonPressHandler);
    return 0;
}
```

3. *Dynamic Plugin System*

Function pointers enable dynamic plugin systems where functions from shared libraries or plugins can be called at runtime. This is useful for applications that need to load and use extensions or plugins.

```
#include <stdio.h>
#include <dlfcn.h>

typedef void (*PluginFunction)();
int main() {
    void *handle;
    PluginFunction pluginFunc;
    char *error;
    // Open shared library
    handle = dlopen("./plugin.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }
    // Load the function from the shared library
    pluginFunc = (PluginFunction) dlsym(handle, "pluginFunction");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        return 1;
```

```
    }
    // Call the function
    pluginFunc();
    // Close the shared library
    dlclose(handle);
    return 0;
}
```

3. Explain how the shared library uses function pointers to link shared library.

Step 1: Create the Shared Library

Create the shared library source file:

Library.c

```
#include <stdio.h>
// Function to add two numbers
int add(int a, int b)
{
    return a + b;
}
```

Compile the shared library:gcc -fPIC -shared -o libsum.so library.c

Step 2: Create the Header File

Library.h

```
// sum_library.h
#ifndef SUM_LIBRARY_H
#define SUM_LIBRARY_H

// Function prototype
int add(int a, int b);
#endif
```

Step 3: Create the Main Program

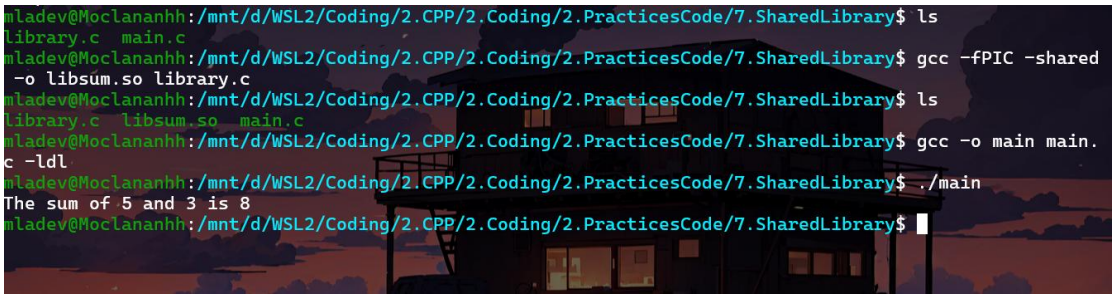
```
// main.c
#include <stdio.h>
#include <dlfcn.h>
#include "library.h"

int main()
{
    void *handle;
    int (*add)(int, int);
    char *error;
    // Open the shared library
    handle = dlopen("./libsum.so", RTLD_LAZY);
    if (!handle)
    {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }
    // Load the 'add' function from the shared library
    *(void **)(&add) = dlsym(handle, "add");
    if ((error = dlerror()) != NULL)
    {
        fprintf(stderr, "%s\n", error);
        return 1;
    }
    // Call the 'add' function using the function pointer
    int result = add(5, 3);
```

```
    printf("The sum of 5 and 3 is %d\n", result);
    // Close the shared library
    dlclose(handle);
    return 0;
}
```

Step 4: Compile the Main Program
gcc -o main main.c -ldl
./main

Output:



3.what is interrupt and inerrupt service routine, how isr will be handled

Interrupt: An interrupt is a signal sent to the CPU by hardware or software indicating that an event needs immediate attention. When an interrupt occurs, the CPU temporarily halts its current operations and executes a function called the Interrupt Service Routine (ISR) to address the event.

Interrupt Service Routine (ISR): An Interrupt Service Routine (ISR) is a special block of code or function that is executed in response to an interrupt. The ISR is responsible for handling the event that caused the interrupt, such as reading data from an I/O device, processing it, or acknowledging that the event occurred.

How ISRs are Handled

1. **Interrupt Occurrence**:An interrupt can be triggered by hardware (e.g., pressing a key on the keyboard, receiving data on a serial port) or by software (e.g., a software-generated interrupt).
2. **Interrupt Acknowledgment**:The CPU acknowledges the interrupt signal and suspends its current execution by saving the current state, such as the program counter (which points to the next instruction to be executed) and CPU registers, onto the stack.
3. **Context Switch**:After saving the current state, the CPU jumps to a predefined memory address where the ISR for that particular interrupt is located. This process is known as a context switch.
4. **ISR Execution**:The ISR executes, handling the interrupt. This may involve reading/writing data, signaling other processes, or other necessary actions related to the interrupt.
5. **Restoring Context**:Once the ISR has completed its task, the CPU restores the saved state (program counter, CPU registers) from the stack, effectively resuming the execution of the program that was interrupted.
6. **Returning to Normal Execution**:The CPU resumes the execution of the interrupted program as if the interrupt had never occurred, continuing from where it left off.

4. Understand the below program

```
include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
typedef struct
{
char int_name[32];
int (*isr)();
}isr_t;

int keyboard_interrupt()
{
printf("Generating keyboard_interrupt..\n");
}

int mouse_interrupt()
{
printf("mouse_interrupt...\n");
}

int rtc_interrupt()
{
printf("rtc_interrupt...\n");
}

int i2c_interrupt()
{
printf("i2c_interrupt..\n");
}

int usb_interrupt()
{
printf("usb_interrupt...\n");
}
isr_t ivt[] =
{
{"isr0", keyboard_interrupt},
{"isr1", mouse_interrupt},
{"isr2", rtc_interrupt},
{"isr3", i2c_interrupt},
{"isr4", usb_interrupt},
};

int main()
{
while(1)
{
printf("Happy \"days\" all\n");

(ivt[rand()%5].isr)();
sleep(1);
}
}
```

This program simulates interrupt handling by using an interrupt vector table containing function pointers. Each iteration of the program randomly selects an interrupt and executes the ISR function corresponding to that interrupt. This helps illustrate how interrupts might be handled in practice.