**ACCESSING I/O DEVICES:**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

// Interrupt handler function for Device 1

void device1\_interrupt\_handler(int signal) {

printf("Device 1 Interrupt Handler\n");

// Handle Device 1 interrupt

}

// Interrupt handler function for Device 2

void device2\_interrupt\_handler(int signal) {

printf("Device 2 Interrupt Handler\n");

// Handle Device 2 interrupt

}

// Enable interrupt for a device

void enable\_interrupt(int device) {

switch(device) {

case 1:

signal(SIGINT, device1\_interrupt\_handler);

break;

case 2:

signal(SIGINT, device2\_interrupt\_handler);

break;

default:

printf("Invalid device\n");

break;

}

}

// Disable interrupt for a device

void disable\_interrupt(int device) {

signal(SIGINT, SIG\_DFL);

}

int main() {

// Enable interrupts for both devices

enable\_interrupt(1);

enable\_interrupt(2);

printf("Interrupts enabled for both devices. Press Ctrl+C to trigger interrupts.\n");

// Loop indefinitely to simulate device operations

while (1) {

// Perform other tasks or wait for interrupts

}

// Disable interrupts before exiting

disable\_interrupt(1);

disable\_interrupt(2);

return 0;

}

**OUTPUT:**

Interrupts enabled for both devices. Press Ctrl+C to trigger interrupts.

Device 2 Interrupt Handler

**BUS ARBITRATION:**

#include <stdio.h>

#define NUM\_DEVICES 3

// Function to simulate bus arbitration using round-robin

int busArbitrationRoundRobin(int currentDevice) {

return (currentDevice + 1) % NUM\_DEVICES;

}

int main() {

int currentDevice = 0; // Start with the first device

printf("Bus Arbitration Simulation\n");

// Simulate 10 cycles of bus access

for (int cycle = 1; cycle <= 10; cycle++) {

printf("Cycle %d: Device %d accesses the bus\n", cycle, currentDevice);

// Perform some operation or task for the device accessing the bus

// ...

// Update the current device using round-robin arbitration

currentDevice = busArbitrationRoundRobin(currentDevice);

// Print a newline for better readability

printf("\n");

}

return 0;

}

**OUTPUT:**

/tmp/xEQvQsC0hU.o

Bus Arbitration Simulation

Cycle 1: Device 0 accesses the bus

Cycle 2: Device 1 accesses the bus

Cycle 3: Device 2 accesses the bus

Cycle 4: Device 0 accesses the bus

Cycle 5: Device 1 accesses the bus

Cycle 6: Device 2 accesses the bus

Cycle 7: Device 0 accesses the bus

Cycle 8: Device 1 accesses the bus

Cycle 9: Device 2 accesses the bus

Cycle 10: Device 0 accesses the bus

**Direct Memory Addressing:**

#include <stdio.h>

int main() {

int variable = 42;

int \*pointer = &variable;

printf("Original value at memory location: %d\n", \*pointer);

// Modify the value at the memory location directly

\*pointer = 99;

printf("Modified value at memory location: %d\n", \*pointer);

return 0;

}

**OUTPUT:**

/tmp/xEQvQsC0hU.o

Original value at memory location: 42

Modified value at memory location: 99

**PIPELINE INTERRUPTS:**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

// Signal handler function

void handleInterrupt(int signum) {

printf("Received interrupt signal (Ctrl+C) - exiting\n");

// You can perform cleanup or additional actions here

\_exit(0); // Terminate the program immediately

}

int main() {

// Set up the signal handler

if (signal(SIGINT, handleInterrupt) == SIG\_ERR) {

perror("Error setting up signal handler");

return 1;

}

// Run an infinite loop

while (1) {

printf("Program is running...\n");

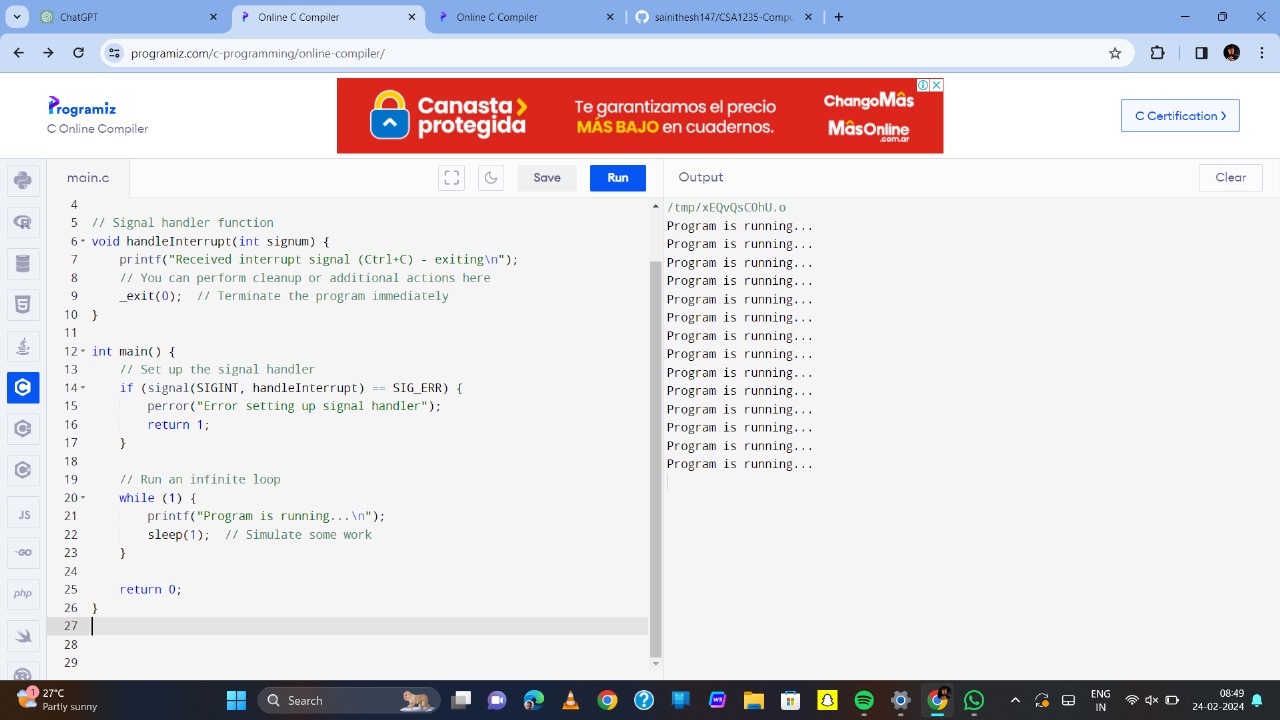
sleep(1); // Simulate some work

}

return 0;

}

**OUTPUT:**

****

**PCI INTERRUPTS:**

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

// Define a signal handler function for the interrupt

void handle\_interrupt(int signum) {

printf("PCI Interrupt Received! Signal Number: %d\n", signum);

}

int main() {

// Install the signal handler for SIGINT (you might need to use a different signal)

if (signal(SIGINT, handle\_interrupt) == SIG\_ERR) {

perror("Error setting up signal handler");

exit(EXIT\_FAILURE);

}

// Simulate some work in the main loop

while (1) {

printf("Main Loop: Working...\n");

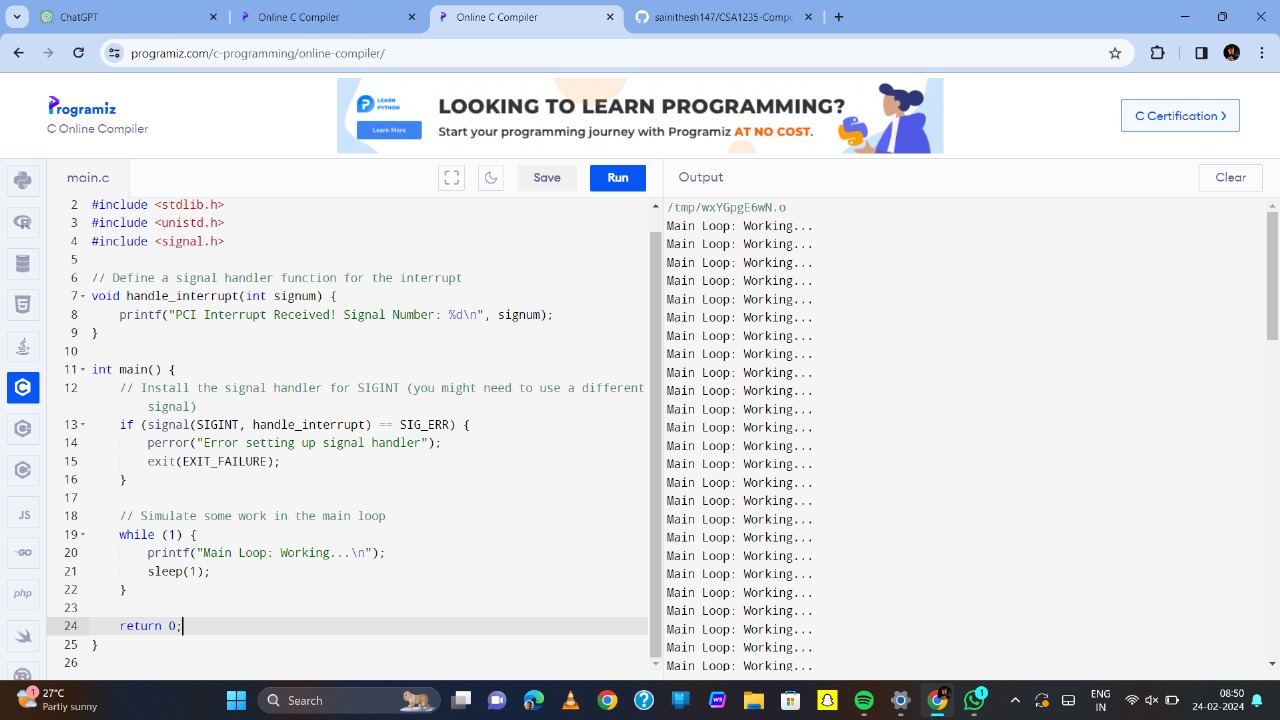
sleep(1);

}

return 0;

}

**Output:**

****

Exceptions vectored interrupts:

#include <stdio.h>

#include <signal.h>

#include <stdlib.h>

// Define a signal handler function for the segmentation fault

void handle\_segfault(int signum) {

printf("Segmentation Fault (Signal %d) - Exiting program\n", signum);

exit(signum);

}

int main() {

// Install the signal handler for SIGSEGV (Segmentation Fault)

if (signal(SIGSEGV, handle\_segfault) == SIG\_ERR) {

perror("Error setting up signal handler");

exit(EXIT\_FAILURE);

}

// Access an invalid memory address to trigger a segmentation fault

int \*ptr = NULL;

\*ptr = 10; // This will cause a segmentation fault

// The program will not reach this point if a segmentation fault occurs

printf("This line will not be executed due to the segmentation fault.\n");

return 0;

}

OUTPUT:

/tmp/xEQvQsC0hU.o

Segmentation Fault (Signal 11) - Exiting program

Controlling Device Request:

#include <stdio.h>

// Simulated device status

int deviceStatus = 0;

// Function to handle device requests

void handleDeviceRequest(int request) {

switch (request) {

case 1:

// Perform action for request 1

printf("Handling Request 1: Turning ON the device.\n");

deviceStatus = 1;

break;

case 2:

// Perform action for request 2

printf("Handling Request 2: Turning OFF the device.\n");

deviceStatus = 0;

break;

default:

printf("Unknown request. Ignoring.\n");

}

}

int main() {

int userRequest;

while (1) {

// Simulating user input or external requests

printf("Enter device request (1: Turn ON, 2: Turn OFF, 0: Exit): ");

scanf("%d", &userRequest);

if (userRequest == 0) {

printf("Exiting program.\n");

break;

}

// Handle the device request

handleDeviceRequest(userRequest);

// Display the current device status

printf("Device Status: %s\n", (deviceStatus == 1) ? "ON" : "OFF");

}

return 0;

}

**OUTPUT:**

/tmp/xEQvQsC0hU.o

Enter device request (1: Turn ON, 2: Turn OFF, 0: Exit):