**Creating a full-fledged cache simulation in a C program involves significant complexity. However, I can provide a simplified example that demonstrates the basic concepts of a cache.**

#include <stdio.h>

#include <stdlib.h>

#define CACHE\_SIZE 4

#define MAIN\_MEMORY\_SIZE 16

// Structure to represent a cache block

typedef struct {

int valid; // Valid bit to indicate whether the cache block contains valid data

int tag; // Tag bits for identifying the memory block

int data; // Data stored in the cache block

} CacheBlock;

// Function to read data from cache

int readFromCache(int address, CacheBlock \*cache) {

int cacheIndex = address % CACHE\_SIZE; // Calculate cache index

int tag = address / CACHE\_SIZE; // Calculate tag bits

// Check if cache block is valid and contains the required data

if (cache[cacheIndex].valid && cache[cacheIndex].tag == tag) {

printf("Cache hit! Data found in cache. Data: %d\n", cache[cacheIndex].data);

return cache[cacheIndex].data;

} else {

printf("Cache miss! Data not found in cache.\n");

return -1; // Indicate cache miss

}

}

// Function to write data to cache

void writeToCache(int address, int data, CacheBlock \*cache) {

int cacheIndex = address % CACHE\_SIZE; // Calculate cache index

int tag = address / CACHE\_SIZE; // Calculate tag bits

// Update cache block with new data

cache[cacheIndex].valid = 1;

cache[cacheIndex].tag = tag;

cache[cacheIndex].data = data;

printf("Data written to cache. Address: %d, Data: %d\n", address, data);

}

int main() {

CacheBlock cache[CACHE\_SIZE]; // Cache memory

// Initialize cache memory

for (int i = 0; i < CACHE\_SIZE; i++) {

cache[i].valid = 0;

cache[i].tag = 0;

cache[i].data = 0;

}

// Simulate cache read and write operations

int address, data;

// Perform cache read operation

printf("Enter memory address to read from cache: ");

scanf("%d", &address);

int readData = readFromCache(address, cache);

if (readData == -1) {

printf("Data not found in cache. Reading from main memory.\n");

// Simulate reading from main memory (not implemented in this example)

}

// Perform cache write operation

printf("Enter memory address and data to write to cache: ");

scanf("%d %d", &address, &data);

writeToCache(address, data, cache);

return 0;

}

**Develop a C program to simulate Virtual Memory operations. The objective is to create a simulation where the program emulates the behavior of a virtual memory system, including address translation, page faults, and paging strategies**

A simple C program that simulates virtual memory operations, including address translation, page faults, and paging strategies:

#include <stdio.h>

#include <stdlib.h>

#define PAGE\_SIZE 1024

#define NUM\_PAGES 256

#define MEMORY\_SIZE (PAGE\_SIZE \* NUM\_PAGES)

#define FRAME\_SIZE 256

#define NUM\_FRAMES (MEMORY\_SIZE / FRAME\_SIZE)

int page\_table[NUM\_PAGES];

char physical\_memory[MEMORY\_SIZE];

char disk\_space[MEMORY\_SIZE];

void init\_page\_table() {

for (int i = 0; i < NUM\_PAGES; i++) {

page\_table[i] = -1; // Indicates page is not in memory

}

}

void handle\_page\_fault(int page\_number, int frame\_number) {

printf("Page fault: Page %d is not in memory. Loading from disk to frame %d.\n", page\_number, frame\_number);

// Simulate loading page from disk into memory

for (int i = 0; i < PAGE\_SIZE; i++) {

physical\_memory[frame\_number \* PAGE\_SIZE + i] = disk\_space[page\_number \* PAGE\_SIZE + i];

}

page\_table[page\_number] = frame\_number;

}

void read\_memory(int virtual\_address) {

int page\_number = virtual\_address / PAGE\_SIZE;

int offset = virtual\_address % PAGE\_SIZE;

if (page\_table[page\_number] == -1) {

int free\_frame = rand() % NUM\_FRAMES;

handle\_page\_fault(page\_number, free\_frame);

}

int physical\_address = page\_table[page\_number] \* PAGE\_SIZE + offset;

char value = physical\_memory[physical\_address];

printf("Read from virtual address %d. Physical address: %d. Value: %c\n", virtual\_address, physical\_address, value);

}

int main() {

// Initialize page table

init\_page\_table();

// Initialize disk space with random data

for (int i = 0; i < MEMORY\_SIZE; i++) {

disk\_space[i] = rand() % 256; // Random byte

}

// Perform some memory reads

read\_memory(0); // Page 0

read\_memory(5000); // Page 4

read\_memory(4096); // Page 4

read\_memory(8192); // Page 8

return 0;

}

**C program to demonstrate Direct Memory Access (DMA) for accessing two I/O devices:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdint.h>

// Define memory addresses for sensor and actuator data

#define SENSOR\_START\_ADDR 0x1000

#define SENSOR\_END\_ADDR 0x1FFF

#define ACTUATOR\_START\_ADDR 0x2000

#define ACTUATOR\_END\_ADDR 0x2FFF

// Simulated sensor and actuator data

volatile uint8\_t sensor\_data = 0;

volatile uint8\_t actuator\_data = 0;

// Function to simulate sensor data generation

void generate\_sensor\_data() {

while (1) {

// Simulate sensor generating data

sensor\_data = rand() % 256; // Random value between 0 and 255

usleep(100000); // Sleep for 100 milliseconds

}

}

// Function to simulate actuator data processing

void process\_actuator\_data() {

while (1) {

// Simulate actuator processing data

printf("Actuator received command: %d\n", actuator\_data);

usleep(200000); // Sleep for 200 milliseconds

}

}

// Function to perform DMA transfer from sensor data to main memory

void sensor\_dma\_transfer() {

// Simulate DMA transfer from sensor data to memory

for (uint32\_t addr = SENSOR\_START\_ADDR; addr <= SENSOR\_END\_ADDR; addr++) {

// Read sensor data and write to memory address

// Actual DMA transfer logic would be hardware-dependent

// Here, we simply copy sensor\_data to memory location

\*((uint8\_t\*)addr) = sensor\_data;

}

}

// Function to perform DMA transfer from main memory to actuator data

void actuator\_dma\_transfer() {

// Simulate DMA transfer from memory to actuator data

for (uint32\_t addr = ACTUATOR\_START\_ADDR; addr <= ACTUATOR\_END\_ADDR; addr++) {

// Read data from memory address and assign to actuator

// Actual DMA transfer logic would be hardware-dependent

// Here, we simply copy data from memory to actuator\_data

actuator\_data = \*((uint8\_t\*)addr);

}

}

int main() {

// Initialize random number generator

srand(time(NULL));

// Create threads for sensor data generation and actuator data processing

pthread\_t sensor\_thread, actuator\_thread;

pthread\_create(&sensor\_thread, NULL, (void\*)generate\_sensor\_data, NULL);

pthread\_create(&actuator\_thread, NULL, (void\*)process\_actuator\_data, NULL);

// Simulate DMA transfers between sensor, memory, and actuator

while (1) {

// Perform DMA transfer from sensor to memory

sensor\_dma\_transfer();

// Perform DMA transfer from memory to actuator

actuator\_dma\_transfer();

}

return 0;

}

**Direct Memory Access (DMA) is a method that allows hardware devices to transfer data directly to or from memory without involving the CPU. Here's a simple C program that simulates a DMA transfer:**

Copy code

#include <stdio.h>

#include <stdlib.h>

#define MEMORY\_SIZE 1024

// Function to simulate DMA transfer

void dmaTransfer(int \*memory, int source[], int length, int destination) {

// Check if the destination address is within the memory bounds

if (destination < 0 || destination + length > MEMORY\_SIZE) {

printf("Error: Destination address out of bounds.\n");

return;

}

// Copy data from source to memory

for (int i = 0; i < length; i++) {

memory[destination + i] = source[i];

}

printf("DMA transfer completed successfully.\n");

}

int main() {

int memory[MEMORY\_SIZE] = {0}; // Initialize memory with zeros

int source[] = {1, 2, 3, 4, 5}; // Sample data to transfer

int destination;

// Prompt user to specify destination address

printf("Enter the destination address for DMA transfer (0 - %d): ", MEMORY\_SIZE - 1);

scanf("%d", &destination);

// Perform DMA transfer

dmaTransfer(memory, source, sizeof(source) / sizeof(source[0]), destination);

// Print memory contents after DMA transfer

printf("Memory contents after DMA transfer:\n");

for (int i = 0; i < MEMORY\_SIZE; i++) {

printf("%d ", memory[i]);

}

printf("\n");

return 0;

}

**Simulating PCI interrupts** in a C program requires interfacing with hardware, which is typically done using operating system APIs or specialized libraries. Below is a simplified example that illustrates handling interrupts using signal handlers in a Unix-like environment. Please note that this example may not directly relate to PCI interrupts, but it demonstrates the concept of interrupt handling.

Copy code

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

// Signal handler function for handling interrupts

void interruptHandler(int signum) {

printf("PCI interrupt received. Processing...\n");

// Add your interrupt handling code here

}

int main() {

// Register the signal handler for SIGINT (Ctrl+C)

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

perror("Error registering signal handler");

return 1;

}

printf("Waiting for PCI interrupt (Press Ctrl+C to simulate)...\n");

// Keep the program running to wait for interrupts

while(1) {

sleep(1); // Sleep for 1 second

}

return 0;

}

**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:**

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

28. RESTORING DIVISION

#include <stdio.h>

void RestoringDivision(int dividend, int divisor) {

int quotient = 0;

int remainder = 0;

for (int i = 0; i < 32; i++) { // Assuming 32-bit integers

remainder = (remainder << 1) | ((dividend >> 31) & 1);

dividend = dividend << 1;

remainder -= divisor;

if (remainder >= 0) {

quotient = (quotient << 1) | 1;

} else {

quotient = quotient << 1;

remainder += divisor;

}

}

printf("Quotient: %d\n", quotient);

printf("Remainder: %d\n", remainder);

}

int main() {

int dividend, divisor;

printf("Enter the dividend: ");

scanf("%d", &dividend);

printf("Enter the divisor: ");

scanf("%d", &divisor);

RestoringDivision(dividend, divisor);

return 0;

}

37. HAZARDS

#include <stdio.h>

int main() {

int a = 5, b = 10, c = 15, result;

// Data Hazard - Read After Write (RAW)

result = a + b; // Data dependency on 'a' and 'b'

printf("Data Hazard (RAW): %d\n", result);

// Control Hazard - Conditional Branch

if (c > 10) {

result = a + b;

} else {

result = a - b;

}

printf("Control Hazard: %d\n", result);

// Structural Hazard - Resource Conflict

int array1[5], array2[5];

for (int i = 0; i < 5; i++) {

array1[i] = i;

}

// Structural Hazard: Both loops trying to access the array simultaneously

for (int i = 0; i < 5; i++) {

array2[i] = array1[i] \* 2;

}

printf("Structural Hazard: Array values multiplied by 2.\n");

return 0;

}

BOOTH ALGORITHM

#include <stdio.h>

int boothMultiplication(int multiplicand, int multiplier) {

int result = 0;

int a = multiplicand;

int b = multiplier;

int product = 0;

int count = 4;

while (count != 0) {

if (b & 1) {

product += a;

}

a <<= 1;

b >>= 1;

count--;

}

result = product;

return result;

}

int main() {

int multiplicand = 5; // Binary: 0101

int multiplier = 3; // Binary: 0011

int result = boothMultiplication(multiplicand, multiplier);

printf("Result of multiplication: %d\n", result);

return 0;

}

C program For Cache hit and cache miss

#include <stdio.h>

#include <stdlib.h>

#define CACHE\_SIZE 4

typedef struct {

int valid;

int tag;

// Add other necessary fields for cache, like data if needed

} CacheLine;

int cacheHits = 0;

int cacheMisses = 0;

// Simulate a direct-mapped cache

CacheLine cache[CACHE\_SIZE];

void initializeCache() {

for (int i = 0; i < CACHE\_SIZE; i++) {

cache[i].valid = 0;

cache[i].tag = -1;

}

}

void accessMemory(int address) {

int index = address % CACHE\_SIZE;

int tag = address / CACHE\_SIZE;

if (cache[index].valid && cache[index].tag == tag) {

// Cache hit

cacheHits++;

printf("Cache Hit!\n");

} else {

// Cache miss

cacheMisses++;

printf("Cache Miss!\n");

// Update the cache with the new data

cache[index].valid = 1;

cache[index].tag = tag;

// Fetch data from main memory if needed

// ...

}

}

int main() {

initializeCache();

// Access some memory addresses for demonstration

accessMemory(1024);

accessMemory(512);

accessMemory(1024); // This should be a cache hit

accessMemory(2048);

printf("Cache Hits: %d\n", cacheHits);

printf("Cache Misses: %d\n", cacheMisses);

return 0;

}

C program for ROM

#include <stdio.h>

#include <stdlib.h>

#include <time.h> // Include time.h for seeding the random number generator

#define ROM\_SIZE 8192 // Size of ROM in bytes

// Function to read data from ROM

char readFromROM(const char ROM[], int address) {

// Check if the address is within the ROM size

if (address >= 0 && address < ROM\_SIZE) {

return ROM[address]; // Return the data at the specified address in ROM

} else {

printf("Error: Invalid address.\n");

return -1; // Return an error code if the address is out of bounds

}

}

int main() {

// Allocate and initialize ROM

char ROM = (char)malloc(ROM\_SIZE);

if (ROM == NULL) {

printf("Error: Unable to allocate memory for ROM.\n");

return 1;

}

// Seed the random number generatorrt

// Initialize ROM with random data

for (int i = 0; i < ROM\_SIZE; i++) {

ROM[i] = rand() ; // Fill ROM with random data (0 to 255)

}

// Simulate reading data from ROM

int address;

printf("Enter address to read from ROM (-1 to exit): ");

while (scanf("%d", &address) && address != -1) {

char data = readFromROM(ROM, address);

printf("Data read from ROM at address %d: %d\n", address, data);

printf("Enter address to read from ROM (-1 to exit): ");

}

// Free allocated memory

free(ROM);

return 0;

}

C program for RAM

#include <stdio.h>

#include <stdlib.h>

#include <time.h> // Include time.h for seeding the random number generator

#define RAM\_SIZE 8192 // Size of RAM in bytes

// Function to read data from RAM

char readFromRAM(const char RAM[], int address) {

// Check if the address is within the RAM size

if (address >= 0 && address < RAM\_SIZE) {

return RAM[address]; // Return the data at the specified address in RAM

} else {

printf("Error: Invalid address.\n");

return -1; // Return an error code if the address is out of bounds

}

}

// Function to write data to RAM

void writeToRAM(char RAM[], int address, char data) {

// Check if the address is within the RAM size

if (address >= 0 && address < RAM\_SIZE) {

RAM[address] = data; // Write data to the specified address in RAM

} else {

printf("Error: Invalid address.\n");

}

}

int main() {

// Allocate and initialize RAM

char RAM = (char)malloc(RAM\_SIZE);

if (RAM == NULL) {

printf("Error: Unable to allocate memory for RAM.\n");

return 1;

}

// Seed the random number generator

// Initialize RAM with random data

for (int i = 0; i < RAM\_SIZE; i++) {

RAM[i] = rand(); // Fill RAM with random data (0 to 255)

}

// Simulate reading and writing data to RAM

int address;

char data;

printf("Enter address to read from RAM (-1 to exit): ");

while (scanf("%d", &address) && address != -1) {

char dataFromRAM = readFromRAM(RAM, address);

printf("Data read from RAM at address %d: %d\n", address, dataFromRAM);

printf("Enter data to write to RAM: ");

scanf("%d", &data);

writeToRAM(RAM, address, data); // Write data to RAM

printf("Data written to RAM at address %d: %d\n", address, data);

printf("Enter address to read from RAM (-1 to exit): ");

}

// Free allocated memory

free(RAM);

return 0;

}